ZEUS-3D USER MANUAL

Version 3.5

David A. Clarke
Institute for Computational Astrophysics
Saint Mary's University
Halifax NS, Canada B3H 3C3
`http://www.ica.smu.ca/zeus3d`

October, 2007

Contents

Preface

Most, if not all of the astrophysical MHD codes used around the world bearing the name *ZEUS* can trace their roots to the original 2-D code developed by Michael Norman and myself in 1986. Of these, this manual describes a version of the code that I have been developing and maintaining ever since. The interested reader will find a complete history of this code from its inception to the present release in the "history deck" of the source code.

The pervasiveness of *ZEUS* throughout the world is in large part due to the generous spirit of Michael Norman whose vision included "astrophysical community codes" to serve theorists much like *AIPS* serves radio astronomers. *ZEUS-3D* was first developed at the National Center for Supercomputing Applications (NCSA) between 1988 and 1990, and in 1992 version 3.2 was made available to the public. A few years later, MLN released the MPI version of the code (*ZEUSMP*) and use of the code spread. Many versions of the code now exist and have been significantly modified by various users to perform simulations from comet-planet collisions to cosmology.

While the *ZEUS*-family of codes are not fully-upwinded (like a Godunov scheme, for example), they have proven to be flexible and robust. One can add almost any physical process to the code without worrying too much about its effects on the underlying MHD scheme. It has therefore found a niche amongst numerically literate, though perhaps not expert, astrophysicists who have a computational problem to investigate but neither the time nor the resources to develop their own code.

One of the roles of the recently formed Institute for Computational Astrophysics (ICA) at Saint Mary's University is to provide and in some cases support code to the astrophysical community. To this end, the ICA web page (`http://www.ica.smu.ca/zeus3d`) now places into the public domain this author's double-precision version of *ZEUS-3D* (version 3.5; `dzeus35`). This code is distinct from the NCSA/UCSD/LCA version of the code (*ZEUSMP*; `http://lca.ucsd.edu`) which has not enjoyed significant algorithm development in more than a decade. As evident in the §1, version 3.5 has undergone signifcant code development and bears little resemblence algorithmically to version 3.2 released more than 15 years ago and, for that matter, to *ZEUSMP*. This includes, for example, a "planar-split" MHD algorithm, fully conservative in 1-D, self-consistent boundary conditions, an "N–log N" Poisson solver, and a full suite of graphics. Future releases will be fully conservative in 3-D and second order accurate in both time and space as operator splitting is abandonned. The code now runs under *openMP* with 98% parallelism [yielding a speed-up factor of 12.5 (20) with 16 (32) processors] and AZEuS[1], an *AMR* version of the code and still under development at the time of this writing, will become available from a parallel site (`http://www.ica.smu.ca/azeus`) in the near future.

Conditions for use of this code are on the next page. The proper citation for referencing the algorithms used in `dzeus35` is:

Clarke, D. A., *A Consistent Method of Characteristics for Multidimensional MHD*, 1996, ApJ, 457, 291.

It is requested that any publication reporting results performed by `dzeus35` or any of its

[1] Adaptive Zone Eulerian Scheme

derivatives include the following acknowledgement:

> Use of *ZEUS-3D*, developed by D. A. Clarke at the Institute for Computational Astrophysics (`www.ica.smu.ca`) with financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC), is hereby acknowledged.

If length is an issue, the following will also suffice:

> Use of *ZEUS-3D*, developed by D. Clarke at the ICA (`www.ica.smu.ca`) with support from NSERC, is acknowledged.

Inquiries about this software, constructive criticism, bug reports, *etc.*, should be directed to the *ZEUS* forum, accessible on line at `http://www.ica.smu.ca/zeus3d`.

David Clarke, October, 2007
Institute for Computational Astrophysics
Saint Mary's University
Halifax, NS, Canada B3H 3C3

Implicit user agreement

In what follows, *this software* refers to "*ZEUS-3D*, version 3.5" (`dzeus35`), and *the author* refers to David A. Clarke, ICA, Halifax. It is assumed that anyone using this code has read, understood, and agreed to the following conditions of use:

1. Distribution of this software shall remain the purview of the author. A user is free to share this software with students and co-workers, but requests from those not working directly with the user should be directed to `www.ica.smu.ca/zeus3d`.

2. This software shall be used exclusively for education, research, non-profit, and non-military purposes. Specific written permission from the author must be obtained before any commercial use of this software is undertaken.

3. The banner and history decks (first two modules of the source code) shall remain with this software and any descendent developed from and still based substantially upon this software.

4. The name(s) of the institution(s) with which the author is or has been affiliated shall not be used to publicise any data and/or results generated by this software. All findings and their interpretations are the opinions of the user and do not necessarily reflect those of the author nor the institution(s) with which the author is or has been affiliated.

The author makes no representations about the suitability of this software for any purpose. Subject to the above conditions, this software and accompanying manual are provided "as is" without expressed or implied warranty.

Acknowledgements

The author wishes to express his gratitude to students, research associates, and collaborators, past and present, for their valuable contributions toward the development of `dzeus35`, and in particular in debugging, providing and/or developing subroutines and algorithms, giving advice, and development of this user manual. In alphabetical order, these include Jack Burns, Mike Casey, Jean Pierre DeVilliers, Kevin Douglas, Phil Hardee, John Hawley, Chris Howard, Byung-Il Jun, Chris Loken, Pierre-Yves Longaretti, Nick MacDonald, Alexander Men'shchikov, Rachid Ouyed, Jon Ramsey, Mark Richardson, Alex Rosen, Jim Stone, Martin Sulkanen, and Joel Tanner.

Acknowledgement is made of the use and incorporation of routines from *Numerical Recipes* by William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. This is an epic tomb of enormous benefit to the computational science community, and the *ZEUS-3D* project has benefited from this classic text on numerous occasions.

The author wishes to thank Kevin Kohler of the Oceanographic Center at Nova Southeastern University (`http://www.nova.edu/ocean/psplot.html`) for his kind permission to make available the source code of *PSPLOT* with `dzeus35`. *PSPLOT* has simplified enormously in-line graphics which had traditionally been accomplished with NCAR graphics.

The author also wishes to thank Professor Tom Jones of the Department of Astronomy at the University of Minnesota for his permission to include his Riemann solver into this release. These modules provide the "analytical" comparator for the suite of 1-D Riemann problems that comprise a significant portion of the test suite used to confirm *ZEUS-3D*.

Over the years, financial and technical support for the *ZEUS* development project(s) has been provided by many sources, including the NCSA and the University of Illinois, the American National Science Foundation and NASA, the Harvard-Smithsonian Center for Astrophysics, Saint Mary's University, and NSERC.

Finally, and most profoundly, the author wishes to thank his former advisor and mentor, Michael Norman, for his vision of a community astrophysics code which came to be known as *ZEUS*. Some of the coding in `dzeus35` still bears Mike's signature, and certainly the fundamental structure of the program follows the Jim Wilson and Mike Norman school of thought.

ZEUS-3D USER MANUAL

Version 3.5, David A. Clarke, ICA, October 2007

1 Introduction

1.1 VERSION 3.0

ZEUS-3D is a 3-D magnetohydrodynamics (MHD) solver, and although it was designed with astrophysical applications in mind, fluid dynamic problems in the other physical sciences can be addressed with this code too. The code is now about 35,000 lines of *FORTRAN* and growing, and represents many years of work by many people. During the past two years, I have been the primary developer of the code, although algorithms and structures developed by many others over the past 20 years have been freely used. These include Philip Colella, Chuck Evans, John Hawley, Michael Norman, Larry Smarr, Jim Stone, Bram van Leer, Jim Wilson, Karl-Heinz Winkler, Paul Woodward, and others.

ZEUS-3D was created as part of the *ZEUS* development project, begun and headed by Dr. Michael Norman at the NCSA (National Center for Supercomputing Applications). It has been Mike's continuing efforts to support this project, both financially and intellectually, that have made the development of *ZEUS-3D* possible. Dr. Jim Stone, also a member of the *ZEUS* development project, was the principle creator of *ZEUS-2D*, the predecessor to *ZEUS-3D*. Although the two codes now differ substantially, the efforts that Jim and Mike made to develop the magnetic field algorithm and the modularity of the code are still very evident in *ZEUS-3D*.

In its present incarnation, *ZEUS-3D* is a three-dimensional ideal (non-resistive, non-viscous, adiabatic) non-relativistic magnetohydrodynamical (MHD) fluid solver which solves the following coupled partial differential equations as a function of time and space:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{1}$$

$$\frac{\partial \mathbf{s}}{\partial t} + \nabla \cdot (\mathbf{s}\ \mathbf{v}) = -\nabla p - \rho \nabla \Phi + \mathbf{J} \times \mathbf{B} \tag{2}$$

$$\frac{\partial e}{\partial t} + \nabla \cdot (e \mathbf{v}) = -p \nabla \cdot \mathbf{v} \tag{3}$$

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{v} \times \mathbf{B}) \tag{4}$$

where:

ρ = matter density
$\mathbf{v}$ = velocity flow field
$\mathbf{s}$ = $\rho \mathbf{v}$ = momentum density vector field
p = thermal pressure

Φ = gravitational potential
$\mathbf{J}$ = current density
$\mathbf{B}$ = magnetic induction
e = internal energy density (per unit volume)

The code possesses the following numerical attributes:

1. finite differencing on an Eulerian mesh (but possibly moving in an average sense with the fluid);
2. fully explicit in time and therefore subject to the CFL limit;
3. operator and directional splitting of the MHD equations;
4. can be used efficiently for 1-D and 2-D simulations with any of the coordinates reduced to symmetry axes;
5. Cartesian geometry for 3-D simulations, Cartesian and cylindrical coordinates for 2-D simulations, Cartesian, cylindrical, and spherical coordinates for 1-D simulations;
6. written in a "covariant" fashion to minimise the effects of the different coordinate systems on the structure of the code;
7. fully staggered grid, with scalars (density and internal energy) zone-centred and vector components (velocity and magnetic field) face-centred [derived vector components (current density and *emf*s) are edge-centred];
8. von-Neumann Richtmyer artificial viscosity to smear shocks;
9. upwinded, monotonic interpolation using one of donor cell (first order), van Leer (second order), or piecewise parabolic interpolation—PPI (third order) algorithms;
10. Consistent Advection used to evolve internal energy and momenta; and
11. Constrainted Transport modified with the Method of Characteristics used to evolve the magnetic fields.

This code is strictly Newtonian. Relativistic astrophysics cannot be simulated in any way with this version. No explicit account for relativistic particles is incorporated either. The code assumes strict charge neutrality at all times—it is not a plasma code. It is assumed that the fluid is thermal, and is coupled to the magnetic fields via collisions with an ionised component which never undergoes charge separation. Pressure is assumed to be isotropic and gravitation is limited to the specification of a point mass. A fully three-dimensional Poisson-solver is planned for the next version (3.1) which will account for the self-gravity of the fluid.

The purpose of this manual is not to educate the potential user on numerical techniques, physical justification of the assumptions inherent to the code, or even what the potential problems to be solved are. Instead, it is assumed that the user is intimately familiar with the fundamentals of MHD and has come up with a complex problem to solve which is completely

described by equations 1 through 4. It is also assumed that the user has a working knowledge of *UNIX*. In this spirit, this manual is designed to instruct the user on the mechanics of using *ZEUS-3D* to solve the equations that pen and paper cannot attempt.

D. Clarke, February 1991

1.2 VERSION 3.2

The code has undergone numerous changes since the release of version 3.0 and has grown to nearly 45,000 lines of *FORTRAN* and more than 160 subroutines. Version 3.1 was never actually released as such, and so there is no corresponding manual. This, then, is the first revision of the user manual. The major differences between versions 3.0 and 3.2 include:

1. Line-of-sight integrations through the data volume for a variety of variables, including the Stokes parameters (see §3.9) are possible in both `XYZ` and `ZRP` coordinates. The *EDITOR* definition `RADIO` (§2.2.1) must be set to invoke this display option.

2. An option has been added to generate time slice plots. The *EDITOR* definition `TIMESL` has been added which now must be set in order to get time slice output.

3. Subroutines peculiar for generating polar pixel dumps (written by Carol Song) have been expunged. *ZEUS-3D* now converts polar slices to Cartesian slices "on the fly" before generating pixel dumps.

4. 2-D *NCAR* graphics have been enhanced with better annotation. Polar contours and vector plots now work properly.

5. An *EDITOR* alias `FINISH` has been added which represents a subroutine called after the main loop of the main program `zeus3d`. This gives the user a slot in which to perform various tasks at the end of the run.

6. The code can be micro-tasked for the Crays. Tests indicate that for typical runs, a real-time speed-up of 3.9 can be achieved with 4 dedicated processors.

7. The code will now run efficiently (*i.e.*, it vectorises) as a uni-tasked process on the Convex. This is done by defining the *EDITOR* definition `CONVEXOS`. Multi-tasking on a Convex using the `-O3` option can be done, but yields a real-time speed-up of only about 2.5 on a four processor machine. For runs on the Crays, `UNICOS` must now be defined.

8. More combinations of dimension and geometry are now known to work. The list now includes Cartesian (`XYZ`) with any two, any one, or no symmetry flag(s) set, cylindrical (`ZRP`) with either `JSYM+KSYM` or `KSYM` set, spherical polar (`RTP`) with either `JSYM+KSYM` or `KSYM` set. Other combinations will be debugged as needed.

9. One can now select an isothermal equation of state. A new *EDITOR* definition `ISO` has been added to take advantage of the reduction in memory and computation requirements for isothermal systems.

10. Yu Zhang (NCSA) has implemented a 3-D self-gravity module using the so-called DADI (Dynamical Alternating Direction Implicit) scheme. The *EDITOR* definition `GRAV` must be set if self-gravity is to be invoked.

11. One now has the choice of solving either the total energy equation or the internal energy equation (the only choice in previous versions). The toggle `itote` has been introduced to the namelist `hycon` to specify which of these formalisms is to be used (Byung-Il Jun, NCSA).

12. Pixel, Voxel and *RADIO* dumps may now be made in *HDF* format. This avoids the cumbersome process of "bracketing" the images, but at the cost of more than four times the disc space requirements.

13. The common blocks have been radically restructured, and the way restart dumps are generated has been overhauled entirely. It is now possible to read a restart dump, for example, that was generated by a compiled version of the code with different *EDITOR* macro settings and different values for the array parameters.

14. Ragged boundaries are no longer available. This feature has been expunged from the code for lack of use and because of the increasing effort necessary to incorporate it into new features. Boundaries must now be regular.

Users of version 3.0 will be happy to note that there are no major changes in the way *ZEUS-3D* is compiled or executed, and the namelist parameters have remained more or less fixed. Still, there are enough subtle changes that it might do the experienced user some good to review these notes before attempting to run a job with this new version. Also note that version 3.2 cannot read restart dumps created by version 3.0, and vice versa.

D. Clarke, August, 1992

1.3 VERSION 3.3

The NCSA, under the auspices of the Laboratory for Computational Astrophysics and the leadership of Dr. Michael Norman, has developed `zeus32` into an MHD-cosmology code and continues to make their code available to the community. Independent of the NCSA effort, I and my co-workers have developed `zeus32` into an CR-MHD (CR $\equiv$ cosmic rays) code (`zeus33`). This manual, therefore, describes the first non-NCSA version of the code and was developed at the Harvard-Smithsonian Center for Astrophysics. This version contains more than 52,000 lines of source code and is the most extensive re-write of the code since version 3.0 was first generated from the 2-D template. Most of the routines in the `PHYSICS` group—including the hydrodynamics—have been rewritten in order to implement the new Consistent Method of Characteristics (CMoC). The CMoC was developed to solve the chronic problem of magnetic field explosions in previous MHD algorithms. While substantive to the code, these changes are mostly transparent to the user. Changes of significance to the user include:

1. The *EDITOR* alias `MOC` has been removed, since the MoC algorithm has been replaced with the CMoC algorithm. The option to use the original CT scheme has also been

eliminated since, unlike MoC, CMoC reduces to the original CT scheme in the weak field limit. The *EDITOR* alias `FASTCMOC` has been added to activate the more efficient version of CMoC for cases where the ratio of the flow and Alfvén velocities is not expected to exceed 10^8 for 64-bit words, and 10^4 for 32-bit words.

2. A two-fluid approximation for a relativistic fluid has been installed (Byung-Il Jun, NCSA). It is turned on by specifying the *EDITOR* macro `TWOFLUID`. The two-fluid approximation takes after Jones & Kaing (1991, ApJ, 363, 499) and can reproduce all of their results. The diffusion coefficient is determined by a subroutine linked with the *EDITOR* alias `DIFFUSION`. The diffusion operator is performed using a time-centred, sub-cycling algorithm which allows the CFL limit to be specified independently of the diffusion time scale.

3. A time-centred subcycling option for the artificial viscosity has been installed and is activated by setting `iscyqq=1` in namelist `hycon`. This renders the CFL limit independent of the viscous time scale. For applications with strong shocks, this can reducing computational time by a factor of 2 or more.

4. An additional option for `ARTIFICIALVISC` has been introduced (`gasdiff`) by Byung-Il Jun. This routine uses ordinary gas diffusion to stabilise shocks. It does so without any excess heating often associated with viscosity, but tends to render the solution very smooth since it is applied everywhere.

5. The variables `iordd`, `iorde`, *etc.* and `istpd`, `istpe`, *etc.* have been expunged. In this release, `iord` and `istp` specify respectively the order of the interpolation algorithm and whether the steepener is to be applied in the third order *PPI* algorithm for all variables.

6. The I/O has been updated with the two-fluid variables. In addition, the conventions used in the various I/O routines have been standardised. In particular, with the exception of *RADIO* variables, virtually all variables available for output in any one I/O routine are available in all. By necessity, the *RADIO* variables remain limited.

7. A "pseudogravity" option has been added. The pseudogravity "holds onto" artificial pressure gradients (*e.g.*, a King atmosphere) much like ordinary gravity was used in `ZEUS04` (the predecessor to *ZEUS-2D*). The pseudogravity is activated by setting the *EDITOR* macro `PSGRAV` which is mutually exclusive with `GRAV`. The pseudo-gravitational potential has the same units as pressure (*i.e.*, ρv^2) rather than the usual units of gravitational potential (*i.e.*, v^2). The pseudo-gravitational acceleration is given by $dv/dt = -(\nabla\phi)/\rho$ and is treated exactly as a pressure in the source term routines. Thus, to "hold onto" an artificial atmosphere in a problem initialisation routine, simply define `PSGRAV` and set `gp(i,j,k) = p(i,j,k)`.

8. Bremsstrahlung emission has been added to the *RADIO* dumps.

9. The code has been generalised to run on SUN SPARCstations. The `EDITOR` macro `SUNOS` must be specified for either *SUNOS* or *SOLARIS* operating systems.

D. Clarke, December 1993

1.4 VERSION 3.4

It has been more than a decade since version 3.3 was completed. Except for one bug found in the CMoC algorithm (to do with density, and described below), it has proven to be an extremely robust algorithm. The main problem with the code is its boundary conditions, and this version has seen numerous rewrites and experimentation with its boundary condition routines, none satisfactory.

The main problem is that as released, version 3.3 could not do something as simple as launch an Alfvén wave from an inflow boundary. As released, version 3.4 can, but at the cost of introducing monopoles at an inflow boundary in some circumstances (such as Ouyed-type jets from problem generator `CORONA`), and numerous patches have been installed to prevent or at least limit these. In particular, inflow/outflow boundary conditions should be used with great caution. On the plus side, with help form Pierre-Yves Longaretti, periodic boundary conditions are now *exact*, with both sides of a periodic grid committing identical machine round-off errors.

Other changes to the code include:

1. The code has been upgraded to double precision, and is now called `dzeus34`. Creating the executable `xdzeus34` now requires linking the double precision libraries: `dnamelist.a` and `dsci01.a`.

2. The problem generator for launching jets from accretion discs (à la Ouyed and Pudritz) has been added (corona). A new *EDITOR* definition `POLYTROPE` has been added if the results of solving the internal energy equation are to be set aside in favour of a strict polytrope ($p = \kappa\rho^\gamma$). This feature should be used with extreme caution as a polytrope is not physically equivalent to an adiabatic equation of state (the former forbidding irreversible processes).

3. The problem generator for Couette type flows (Longaretti) has been added.

4. PSGRAV and GRAV may now be set simultaneously, if needed.

5. Yu Zhang's DADI gravity routines, which never worked properly, have all been expunged and two new Poisson solvers have been installed by A. Men'shchikov: SOR (Successive Over-Relaxation), and FMG (Full Multi-Grid). The algorithm is chosen by setting `gravalg` to 1 or 2 for SOR or FMG. Only SOR is fully debugged.

6. The code is now portable to *AIX* (IBM) and *LINUX*, as well as other flavours of compilers such as *NAG* and *WATCOM*.

7. A bug in the CMoC algorithm was fixed. The original scheme used four-point averages of the density to the location of the *emf* when estimating the characteristic velocities. However, it was found that at steep density gradients, this proved disastrous. A degree of freedom overlooked in the original CMoC implementation was exploited to allow the density to be upwinded too, thus preventing steep gradients from over- or underestimating *emf*s.

8. Kinematic viscosity has been added to the code (constant viscosity only), and is triggered by specifying a non-zero value for "`nu`", a global variable, in namelist `HYCON`. "`nu`" is the kinematic viscosity defined by $\nu = VL/\mathcal{R}$, where $\mathcal{R}$ is the Reynolds number of the flow and L and V are length and velocity scales of the problem.

9. A. Men'shchikov has introduced *PSPLOT*[2] to the plotting library `grfx03.a`. Three namelist parameters (`norpp1`, `norpp2`, `norptsl`) will allow for publication-quality graphics with colour without linking any *NCAR* libraries. Two additional user-creatable libraries `psplot.a` and `noncar.a` must be linked instead for this option to work.

10. The subroutines `CURRENT*` have been replaced with `CURL*`, which compute components of the curl. It is a generalisation that may be used to compute the vorticity as well.

11. Vector potentials are now available in all graphics options (*e.g.*, §B.18, §B.19, §B.20, *etc.*). To allow this, "inverse curl" routines have been added (following Arfken, ed. 5, pp. 73–74) that compute a vector potential from a given magnetic field.

D. Clarke, September, 2005

1.5 VERSION 3.5

Modifications to the code from version 3.4 are numerous and invasive, and needed to correct long-standing problems with boundary conditions and to complete the installation of the total energy equation. The code now has more that 90,000 lines of *FORTRAN* and 350 subroutines.

A self-consistent framework for setting magnetic boundary conditions has been developed and installed in this release. In particular, inflow and outflow boundary conditions, while still not perfect, are now much cleaner than in `dzeus34`, and can be used with some confidence. For the user, the visible consequences are four-fold:

1. A distinction is now made between the *skin* values (*e.g.*, variables on the `i=is` face at the inner-i boundary) and proper *boundary* values (*e.g.*, variables, face-centred or zone-centred, at `i=ism1` and `i=ism2` at the inner-i boundary). For inflow conditions, the user must now set the *skin* values of the transverse (to the boundary normal) components of the *emf* (thus, ε_2 and ε_3 at the inner-i boundary), and the *boundary* values of the transverse magnetic field components (thus, B_2 and B_3 at the inner-i boundary). Note that there are no skin values for the transverse magnetic field and the boundary values of the transverse *emf*s are set directly by a new routine `BVALEMFS` called at the top of `CT`.

2. Arrays containing desired inflow values for the normal (to the boundary) magnetic field components (*e.g.*, `b1iib1`, `b2ijb1`, *etc.*) are no longer available. Instead, the normal magnetic field component is now set by the solenoidal condition and therefore "floats".

[2] with kind permission from its author, Kevin Kohler. Please see *Acknowledgements* for the full citation.

3. Routines `BVALB1`, `BVALB2`, and `BVALB3` used to set magnetic boundary values in previous versions are no longer available. Instead, the user must initialise every element of the magnetic field arrays (`b1`, `b2`, and `b3`) in their problem generator including boundary zones making sure that $\nabla \cdot \vec{B} = 0$ at $t = 0$ everywhere.

4. Boundary type 8, a *selective inflow* boundary condition, can now be set and is suitable for sub-magnetosonic inflow conditions. At a given inflow boundary, as many boundary conditions may be set as there are characteristics pointing into the grid (*e.g.*, see Bogovalov, 1997, A&A, 323, 634). For supermagnetosonic flow, this means all seven variables, namely ρ, p (or e), all three components of the velocity, and the two transverse components of the magnetic field (the normal component being determined by the solenoidal condition). On the other hand, for sub-slowmagnetosonic inflow, only four characteristics point into the grid, and three of the inflow variables must be allowed to "float". A variable floats if one sets the boundary type [*e.g.*, `niib(j,k)`] to 8, and if the inflow array for that variable [*e.g.*, `diib1(j,k)` for density, *etc.*] is set to "`huge`" (a global parameter; see §C.6).

Complete details are given in §B.8. Other changes made to this release include:

1. A new parameter, `iords` (§B.7), now lets one specify the order of interpolation for the scalars (*e.g.*, ρ, e, *etc.*) separately from the vector components, which are still controlled by `iord`. The default value for `iords` is `iord`, whose default is still 2 (van Leer). Note that with the scalars interpolated with PPI (`iords=3`) and the contact steepener activated (`istp=2`), contacts in most 1-D Riemann problems are as steep as fast shocks—two or three zones.

2. Additional interpolation schemes have been partially added (*i.e.*, available for scalars only), most notably a velocity-corrected second-order van Leer scheme (`iord=-2`). See §B.7 for details.

3. The new boundary conditions fix many problems, including a long standing one with 3-D toroidal fields in propagating jets. Previously, it was noted that such a field introduced a strong axial field from the `i=is` skin, and tapering the magnetic profile to zero before the jet radius was necessary to avoid this. 3-D toroidal fields now leave the `i=is` skin perfectly cleanly, with zero (to machine round off errors) B_1 left on the skin itself.

4. Stone's *MOC* and the Hawley-Stone variation, *HSMOC*, have been installed to allow for comparisons among the various algorithms. New *EDITOR* aliases `MOC` and `HSMOC` are used to engage these algorithms (§2.2.1).

5. Parameter `ijkn` is now `ijkx` (`x` ⇒ maximum) and a new parameter `ijkn` (`n` ⇒ minimum) has been added. Confusion between the old and new uses of `ijkn` is minimised as these are now set automatically and no longer need to be set by the user in the script file, `dzeus35.s` (§2.3).

6. Installation of the total energy equation is complete, and is now the default (`itote=1`). This has mitigated numerous changes throughout the code, including removal of consistent advection from the energy equations, and the introduction of a new global variable, `et`, that always contains the total energy density, regardless of which energy equation is being solved. The old energy variable, `e`, and its related inflow variables, *e.g.*, `eiib1`, *etc.*, have been changed to `e1` and `e1iib1`, *etc.*, and always contains the internal energy density, regardless of which energy equation is being solved. See §B.7 for further details.

7. Tom Jones' "Reimann solver" has been included, and is compiled when the *EDITOR* macro `RIEMANN` is defined. Its purpose is strictly to provide the "analytical" solution for the suite of 1-D Riemann test problems, and is controlled via namelist `plt1con` (§3.2).

8. *EDITOR* macro `VECPOT` has been added to allow the vector potential to be used as the primary magnetic variables instead of the magnetic field. In principle, this should cause differences only at the machine round-off level, and other than setting the macro, requires no changes by the user. For example, the user would still initialise the magnetic field components everywhere. With `VECPOT` set, the code would then use the "anticurl" routines, `ACURL*`, `*=1,2,3`, to compute the initial vector potential from the initial magnetic field.

9. Toru Okuda's flux-limited diffusion algorithm for radiation HD has been included, though in an incomplete and untested form. It is activated by setting the *EDITOR* macro `RADIATION`.

10. The code has been tuned for *openMP*, and its directives can be inserted automatically by setting `iutask=2` in *EDITOR*'s input deck `inedit` (part of `dzeus35.s`; see §2.3).

D. Clarke, September, 2007

2 Running *ZEUS-3D*

2.1 Overview

At the time of this writing, *ZEUS-3D* runs under *AIX* (IBM), *CONVEXOS* (Convex), *LINUX*, *LINUXIFC*, *LINUXNAG*, *OS2GNU*, *OS2WATCOM*, *SUNOS* (Sun), *SUNOSGNU*, and *UNICOS* (Cray). This manual is written assuming the user will run the code under *SUNOS* (equivalent to *SOLARIS*), although most differences with other OSs are minor and transparent. Some discussion is given where the differences may be more significant. New users can obtain the file `dzeus35.tar` required to install the code (including complete instructions) from the ICA web site (`www.ica.smu.ca/zeus3d`).

In order to run the code, the user will have to edit two files and must have access to various others. The two files to be edited are `zeus35.mac` and `dzeus35.s`. These are relatively short and painless to edit, and their complete descriptions are included in the next two subsections.

Creating the *ZEUS-3D* executable is achieved by running the `dzeus35.s` script file which is done by typing:

```
csh -v dzeus35.s
```

Running this file performs sequentially the following tasks:

1. retrieves all the files from a user-specified home directory;
2. creates a directory called `dzeus3.5` within the user's current directory to store all the source and object files created during compilation;
3. creates a change deck for `dzeus35` containing preprocessor macros and aliases (`zeus35.mac`, next subsection), and changes to the source code (if any) required for the application (the most common and often the only changes which must be made to the source code are to the parameter statements which set the size of the arrays needed for the run.);
4. fires up the *EDITOR* preprocessor;
5. creates the input deck for the `dzeus35` run; and finally
6. makes the executable `xdzeus35` (using the *UNIX* facility *MAKE*).

A description of the file naming convention is required at this point. *ZEUS-3D* refers in a general way to the package and its capabilities while `dzeus35` is more specific, and is a mnemonic for "double precision *ZEUS-3D*, version 3.5". `zeus35` is the common denominator for the names of the principle files required to create the executable. Thus, the source code itself is `dzeus35`, the script file is `dzeus35.s`, the macro file is `zeus35.mac` (there is no leading "`d`" since no changes were needed in this file during migration to double precision), and the executable is `xdzeus35`. However, to confuse matters, the minor files don't follow this convention. The input deck is `inzeus` and the change deck is `chgzeus`—no "`35`" suffix—and the libraries don't even have ZEUS as part of their names. And so it goes. The bottom

line, though, is that if the only changes to be made to the source code are the values of the parameters which set the array dimensions, then there are only two files to be concerned with: `dzeus35.s` and `zeus35.mac`. The rest is automatic.

2.2 The macro file `zeus35.mac`

Below is an example of a `zeus35.mac` file. A similar file can be downloaded from the ICA web site. It is suggested that this file be copied and used as a general template since all the macros used by `dzeus35` are listed in this example.

```
**=======1=========2=========3==========3=========2=========1=========
**                                                                   **
*****************  CONDITIONAL COMPILATION SWITCHES  *****************
**                                                                   **
**  1) symmetry axes:  ISYM, JSYM, KSYM
**
*define   JSYM, KSYM
**
**  2) geometry:  XYZ, or ZRP, or RTP
**
*define   XYZ
**
**  3) physics:  GRAV, ISO, MHD, POLYTROPE, PSGRAV, RADIATION, TWOFLUID
**
*define   MHD
**
**  4) data output modes:  DISP, HDF, PIX, PLT1D, PLT2D, RADIO, TIMESL
**                         VOX
**
*define   PLT1D
**
**  5) operating system:  AIX, CONVEXOS, LINUX, LINUXIFC, LINUXNAG,
**                        OS2GNU, OS2WATCOM, SUNOS, SUNOSGNU, UNICOS
**
*define   SUNOS
**
**  6) other:  DEBUG, FASTCMOC, HSMOC, MOC, RIEMANN, VECPOT
**
*define   FASTCMOC, RIEMANN
**                                                                   **
************************  MODULE NAME ALIASES  ************************
**                                                                   **
**  The modules "BNDYUPDATE", "SPECIAL", "SPECIALSRC", "SPECIALTRN",
**  "FINISH", "PROBLEM", PROBLEMRESTART", "USERSOURCE", and "USERDUMP"
**  are slots available for user-developed subroutines.
**
*alias    START            mstart
*alias    BNDYUPDATE       empty
*alias    EXTENDGRID       empty
*alias    GRAVITY          empty
*alias    SPECIAL          empty
*alias    SOURCE           srcstep
*alias    SPECIALSRC       empty
*alias    TRANSPORT        trnsprt
*alias    SPECIALTRN       empty
*alias    NEWTIMESTEP      newdt
```

```
*alias    NEWGRID          empty
*alias    DATAOUTPUT       dataio
*alias    FINISH           empty
**
*alias    PROBLEM          shkset
*alias    ATMOSPHERE       empty
*alias    PROBLEMRESTART   empty
*alias    USERSOURCE       empty
*alias    ARTIFICIALVISC   viscous
*alias    DIFFUSION        empty
*alias    USERDUMP         empty
**                                                              **
************************  ERROR CRITERIA ALIASES  ***********************
**                                                              **
*alias    GRAVITYERROR     1.0e-6
*alias    GRIDERROR        1.0e-6
*alias    PDVCOOLERROR     1.0e-6
*alias    NEWVGERROR       1.0e-10
*alias    RADIATIONERROR   1.0e-6
**                                                              **
************************  ITERATION LIMITS ALIASES  ***********************
**                                                              **
*alias    GRAVITYITER      600
*alias    GRIDITER         20
*alias    PDVCOOLITER      20
*alias    NEWVGITER        20
*alias    RADIATIONITER    20
```

These are all preprocessor commands (the preprocessor used here is called *EDITOR*—also developed by the author—and for those familiar with the old Cray OS *CTSS*, it has the "look and feel" of *HISTORIAN*), and become part of the "change deck" `chgzeus` created by the script file `dzeus35.s`, described in the next subsection. A change deck is a file which is merged with the source code during the preprocessing step of `dzeus35.s`. Both the source code and the change deck can contain preprocessor commands which are interpreted, carried out, and then expunged from the code by *EDITOR* before the code is compiled by the *FORTRAN* compiler. All preprocessor commands have an asterisk (*) in column 1. Double asterisks indicate a comment. When the preprocessor has finished, the result is a pure *FORTRAN* source code tailored specifically for the problem to be solved. Therefore, in order to customise the code, it is necessary to set the *EDITOR* "definitions" and "aliases" (generically referred to as "macros") found in `zeus35.mac`.

The combined effect of the macros is two-fold. First, they determine what parts of the code are activated and what parts are ignored. Thus, it is possible to eliminate the computations and the memory requirements necessary to evolve the magnetic fields, for example, by not defining the `MHD` macro [this can be done by "commenting out" (double asterisk) the `*define MHD` statement in the example above]. The preprocessor will then remove all coding peculiar to the magnetic fields including the declarations of the magnetic field arrays during the preprocessing step. The compiler never sees the magnetic stuff, and the executable is streamlined for the hydrodynamical problem. Of course, the original source code is not altered by preprocessing it. Rather, the preprocessor creates a precompiled version of the code and stores each subroutine into its own file (to facilitate *MAKE* and debuggers such as *DBX*) in the directory `dzeus3.5` which is created by the script file `dzeus35.s`. Second, the alias

macros can be used to substitute any character string in the code during the preprocessing step.

This manual discusses only those aspects of the *EDITOR* preprocessor necessary for the user to make changes to the code, compile it, and then execute it. A full account of *EDITOR* is given in the manual, `edit21_man.ps`, found in the `manuals` directory of `dzeus35.tar` from `www.ica.smu.ca/zeus3d`.

2.2.1 The *EDITOR* definitions

A description of the definition macros (called "Conditional Compilation Switches" at the top of the given example of `zeus35.mac` above) follows:

1. The code can be streamlined (optimised) for 1-D and 2-D problems by setting the appropriate symmetry macros. If symmetry along any of the i (x_1), j (x_2), or k (x_3) axes is desired, then set the `ISYM`, `JSYM`, or `KSYM` macros. If the macros are not set and a 1-D or 2-D calculation is initialised by the input deck, *ZEUS-3D* will still carry out the sub 3-D computation correctly, but will do so less efficiently.

2. The geometry is set by setting ONE of the `XYZ` (Cartesian), `ZRP` (cylindrical), or `RTP` (spherical polar) macros. Obviously, these macros are mutually exclusive.

3. Defining `GRAV` *and* setting the *EDITOR* alias `GRAVITY` to `gravity` will turn on the Poisson solver and one of two algorithms (SOR, FMG) will be used to solve the self-gravitational potential. The `ISO` macro should be set if an isothermal equation of state is desired. With `ISO` defined, an isothermal equation of state is presumed and the energy variables are not declared saving both computational time and memory. By setting the `MHD` macro, the algorithm for evolving the magnetic fields is activated. With `MHD` on, additional field arrays are declared and the code peculiar to updating the magnetic field is compiled. `POLYTROPE` forces a strict polytropic equation of state. `PSGRAV` (no longer mutually exclusive with `GRAV`) activates the pseudo-gravity feature used to hold onto artificial atmospheres. The macro `RADIATION` enables the (incomplete) flux-limited diffusion algorithm for RMHD. Defining `TWOFLUID` will activate the arrays and coding necessary to solve the energy equation for the second thermal fluid. Note that partial densities and momenta are not tracked for the second fluid; only partial internal energies (and thus partial pressures). The second fluid may be subjected to diffusion, if desired.

4. The graphics enabled during a run are set by the graphics macros. Set `DISP` for display dumps, set `HDF` for *HDF* dump files, set `PIX` to enable 2-D pixel dumps, set `PLT1D` for 1-D line plots, set `PLT2D` for 2-D contour and/or vector plots, set `RADIO` for *RADIO* dump files, set `TIMESL` for time slice dumps, and set `VOX` for 3-D voxel dumps. As many as these may be set simultaneously as necessary. See §3 for a discussion of the various *ZEUS-3D* dump files.

5. The operating system is defined by setting only one of the macros `AIX` (IBM), `CONVEXOS` (Convex), `SUNOS` (SUN's old operating system; applies to all *SUNOS* or *SOLARIS* systems), `LINUX`, and `UNICOS` (Cray). In addition, the peculiarities of several third-party

compilers are supported and can be invoked by defining one of `LINUXIFC`, `LINUXNAG`, `OS2GNU`, `OS2WATCOM`, and `SUNOSGNU` instead.

6. The `DEBUG` macro turns on portions of the code designed for development and debugging, and will send all sorts of messages to the terminal and may even cause the code to crash. It should be invoked only by developers of the code. The faster CMoC algorithm may be invoked by setting the macro `FASTCMOC`. This macro should be set only if the accuracy of the smallest of the flow and Alfvén speeds is unimportant when it falls below 10^{-8} (10^{-4}) times the largest of the speeds for 64-bit (32-bit) words. Otherwise, the general CMoC algorithm (activated by *not* setting the `FASTCMOC` macro) can handle arbitrarily small Alfvén and/or flow speeds accurately, but at the cost of 25% more computational time. The macros `HSMOC` and `MOC` invoke other MHD algorithms which are available for comparison, but are not recommended for general use. The macro `RIEMANN` is needed if 1-D analytical solutions are to be overlaid with the results of 1-D shock-tube tests, and `VECPOT` forces the use of a version of the induction equation based on the vector potential rather than the magnetic field.

2.2.2 The *EDITOR* aliases

The alias macros allow phrases in the code to be substituted for other phrases during the precompiling step. Thus, "Module Name Aliases" (in the middle of the given example of `zeus35.mac` above) give the user control over what subroutines are called during execution. As an example, in the main program of the source code, there is a statement: `call START` which becomes `call mstart` after preprocessing using the given example of `zeus35.mac`. Note that there is no subroutine called `START` but there is a subroutine in the source code called `mstart`. Thus, the user is free, in principle, to create their own initialisation subroutine to be called instead of `mstart` which can be linked into the code by altering the alias setting for `START` from `mstart` to the name of the user's initialising subroutine. Note that by setting any of the Module Name Aliases to `empty` (a subroutine in `dzeus35` which does nothing but return to the calling routine), a Module Name Alias can be effectively "turned off".

Aliases can also be used to set parameters in various parameter statements scattered throughout the source code. These are the "Error Criteria Aliases" and "Iteration Limits Aliases" at the bottom of the given example of `zeus35.mac` above. Thus the *EDITOR* statement:

```
alias GRAVITYERROR     1.0e-6
```

sets the maximum convergence error in the self-gravity module to 10^{-6}. Somewhere in the code is the statement `parameter ( errmax = GRAVITYERROR )` and the preprocessor makes the substitution. However, the majority of the parameters (array dimensions, for example) are set directly in `dzeus35.s` which is described in the next subsection.

To understand better the descriptions of the "Module Name Aliases" which follow, the reader should examine the flow chart in Appendix 1 (*ZEUS-3D* Skeleton). This is a flow chart of the code, and indicates in which order the Module Name Aliases are called. Some subroutines are charged with reading the input data from the input deck `inzeus`. A description of all the input namelist parameters is given in Appendix 2.

1. `START`: This module is called just once before the computations begin. It should initialise all the variables to be used in the simulation and perform all the initial I/O. Currently, the only choice available for `START` is `mstart`.

2. `BNDYUPDATE`: This module is called at the beginning of each loop and allows inflow boundary conditions to be evolved in time should this be necessary for the simulation. Examples of evolving inflow boundary conditions include helically perturbing the inflow at a jet orifice to break the symmetry (`wiggle`), generating magnetic field at the boundary (`bgen`), or `empty` if no inflow boundary update is desired. The user can, of course, supply a subroutine for this alias. See §5.1 for discussion on how to add a subroutine to the code.

3. `EXTENDGRID`: This module will allow the grid to be extended as a disturbance (shock) propagates into initially quiescent zones. Currently, the only options are `extend` and `empty`. The subroutine `extend` will prevent quiescent zones from being updated until the disturbance comes within five zones, potentially saving significant amounts of computational time. Care should be exercised in its use, however. If the subroutine is unsuccessful in determining when the disturbance gets close to an edge of the current computational domain, the results can be disastrous.

4. `GRAVITY`: This module updates the self-gravitational potential. Currently, the only choices are `empty` and `gravity`. If `gravity` is selected, the user will have to choose a Poisson solver (`grvalg` in namelist `grvcon`), as well as a method to determine boundary values (`giib`, *etc.* in namelist `iib`, *etc.*).

5. `SPECIAL`: This is a simplistic solution to the potentially complex problem of the user desiring to add a whole new type of physics to the code. It assumes that changes do not need to be intertwined into existing modules, which in practise, often will be necessary. The three accompanying "plugs" `SPECIALSRC` (for "special" source terms to be added after the artificial viscous step), `USERSOURCE` (for source terms to be added before the artificial viscous step, and `SPECIALTRN` (for "special" transport terms) allow for some flexibility in installing new physics within the current structure, but this still may not be enough for any type of sophisticated addition. Currently, all four macros are set to `empty`.

6. `SOURCE`: This is the module in which source terms are incorporated. For full dynamics, this should be set to `srcstep` (or the user's module if need be) while for problems of pure advection, this should be set to `empty`.

7. `SPECIALSRC`: See `SPECIAL`.

8. `TRANSPORT`: This is the module for the transport of variables across zone boundaries and should be set to `trnsprt` or to the user's equivalent module. It is unlikely that `empty` should ever be used here.

9. `SPECIALTRN`: See `SPECIAL`.

10. `NEWTIMESTEP`: This module determines how the next time step is computed. Since *ZEUS-3D* is an explicit code, all algorithms should incorporate the CFL limit. Current choices are `newdt` for full (M)HD problems, and `advectdt` for pure advection problems.

11. `DATAOUTPUT`: This module is responsible for data I/O. Setting this macro to `dataio` will cause restart dumps, plot files, pixel dumps, voxel dumps, *HDF* files, display files, *RADIO* dumps, time slice dumps, and any other format as specified by the macro `USERDUMP` to be created at time intervals set by the user (§3). Setting the macro to `empty` will prevent all data I/O—probably not a good idea.

12. `FINISH`: This is a "plug" available to the user to have any user-supplied subroutine called once at the end of execution. It could, for example, be used to generate the final plots of certain variables that the user has been monitoring via another user-supplied subroutine set to `USERDUMP`.

13. `PROBLEM`: This macro is used to link the user-supplied "problem generating" subroutine that initialises all flow variables and boundary values. It is called by the subroutine `setup`, which is called by `mstart` (`START`). Alternately, a number of problem generators for a variety of applications already exist in the source code. In the present example, `PROBLEM` is set to `shkset`, an existing problem generator which initialises the variables for a 1-D Riemann problem ("shock tube"); in this case, the so-called *Brio and Wu problem.*

14. `PROBLEMRESTART`: This macro allows the specifications of the problem to be altered should the job be restarted from a restart dump. Set the macro to `empty` if no alteration of the problem is desired (as, for example, to simply extend the evolution time).

15. `USERSOURCE`: See `SPECIAL`.

16. `ARTIFICIALVISC`: This macro specifies which artificial viscosity algorithm should be used. Current options are `viscous`, which uses the von-Neumann Richtmyer artificial viscosity algorithm, and `gasdiff` which invokes ordinary gas diffusion.

17. `DIFFUSION`: This macro specifies the subroutine to use to compute the diffusion coefficient for the two-fluid model. Currently, the only options are `empty` and `diffco`.

18. `USERDUMP`: See `DATAOUTPUT`.

It is unlikely that the "Error Criteria Aliases" or the "Iteration Limits Aliases" should ever have to be changed.

Finally, in addition to the aliases listed above is module name alias `ATMOSPHERE` called by problem generator `JETINIT` which allows a user to specify their own routine to initialise an atmosphere through which a jet is launched. Existing atmosphere routines include `CLOUD` (to set up a jet-cloud collision) and `KING`, which sets up a King atmosphere.

2.3 The script file `dzeus35.s`

Below is a reproduction of the script file `dzeus35.s` found in the `zeus` directory of the file `dzeus35.tar` downloaded from `www.ica.smu.ca/zeus3d`. It can be run by typing: `csh -v dzeus35.s`.

```
#============= SOURCE FILE TO CREATE THE ZEUS EXECUTABLE ============#
#                                                                     #
#                          M H D S O D . X 1                          #
#                                                                     #
#  itote=0, iscyqq=1: prtime=80.072079, last dt=0.19973, nhy=420      #
#  itote=0, iscyqq=0: prtime=80.066829, last dt=0.16281, nhy=504      #
#  itote=1          : prtime=80.023102, last dt=0.16756, nhy=506      #
#  all: iord=2, iords=3, istp=2, qcon=1.0, qlin=0.2, courno=0.75      #
#                                                                     #
#=====================================> Get files from home directory.
if(! -e xedit21) cp ../editor/xedit21 .
if(! -e dnamelist.a) cp ../nmlst/dnamelist.a .
if(! -e dsci01.a) cp ../sci/dsci01.a .
if(! -e grfx03.a) cp ../grfx/grfx03.a .
if(! -e psplot.a) cp ../grfx/psplot.a .
if(! -e nopsplot.a) cp ../grfx/nopsplot.a .
if(! -e noncar.a) cp ../grfx/noncar.a .
#======================> If necessary, create the directory "dzeus3.5".
if(! -e dzeus3.5) mkdir dzeus3.5
#-----------------------------------------------> Create the change deck.
rm -f chgzeus
cat << EOF > chgzeus
*read zeus35.mac
*d par.42,43
       parameter     (   in = 555,   jn =   1,   kn =   1 )
       parameter     ( nxpx =   1, nypx =   1, nxrd =   1, nyrd =   1 )
**read chguser
EOF
#======================> Create the input deck for EDITOR, and execute.
rm -f inedit
cat << EOF > inedit
 \$editpar   inname='dzeus35'
          , ibanner=0, idump=1, job=3, safety=0.20
          , ipre=1, inmlst=1, iupdate=1, iutask=0
          , chgdk='chgzeus'
          , branch='dzeus3.5'
          , makename='makezeus', xeq='xdzeus35'
          , coptions='-g -C -ftrap=common', loptions='-g'
c         , coptions='-fast -fsimple=1', loptions='-fast'
c         , speccopt='-O1', specdk='corona','phistv','nmlsts','plot1d'
          , libs='checkin.o dnamelist.a dsci01.a grfx03.a psplot.a
  noncar.a'                                                          \$
c         , libs='checkin.o dnamelist.a dsci01.a grfx03.a psplot.a
c noncar.a -lmfhdf -ldf -ljpeg -lz -lX11 -lcgm'                      \$
c         , libs='checkin.o dnamelist.a dsci01.a grfx03.a psplot.a
c -lmfhdf -ldf -ljpeg -lz -lX11 -lcgm -lncarg -lncarg_c -lncarg_gks
c -lncarg_ras -lngmath'                                              \$
EOF
chmod 755 xedit21
./xedit21
#-------------------------------------> Create the input deck for ZEUS.
rm -f inzeus
```

```
cat << EOF > inzeus
 \$iocon    iotty=6, iolog=2                                                \$
 \$rescon   dtdmp=0.0, idtag='xd'                                           \$
 \$ggen1    nbl=550, x1min=0.0,x1max=550., igrid=1, x1rat=1.0, lgrid=.t.\$
 \$ggen2                                                                    \$
 \$ggen3                                                                    \$
 \$pcon     nlim= 999999, tlim=80.0, ttotal=900.0, tsave=10.0               \$
 \$hycon    qcon=1.0, qlin=0.2, courno=0.75, iord=2, iords=3, istp=2
         , itote=1, iscyqq=0                                              \$
 \$iib      niib(1,1)=9                                                     \$
 \$oib      noib(1,1)=9                                                     \$
 \$ijb                                                                      \$
 \$ojb                                                                      \$
 \$ikb                                                                      \$
 \$okb                                                                      \$
 \$grvcon                                                                   \$
 \$eqos     gamma=2.0                                                       \$
 \$gcon                                                                     \$
 \$extcon                                                                   \$
 \$plt1con  iplt1dir=1, dtplt1=80.0, corl=1, aspect=1.0, np1h=2, np1v=2
         , norpp1=2, ip1soln=12*1, xdiscp1=200.0
         , plt1var= 'd ', 'se', 'p ', 'et', 'v1', 'v2', 'v3', 'ma'
                  , 'b1', 'b2', 'b3', 'bd'                                \$
 \$plt2con                                                                  \$
 \$pixcon                                                                   \$
 \$voxcon                                                                   \$
 \$usrcon                                                                   \$
 \$hdfcon                                                                   \$
 \$tslcon                                                                   \$
 \$discon                                                                   \$
 \$radcon                                                                   \$
 \$pgen     idirect=1, n0=200, d0=1.000, e10=1.0, v10=0.0, b10=0.75
         , b20=0.6, b30=0.8                                               \$
 \$pgen     idirect=1, n0=350, d0=0.125, e10=0.1, v10=0.0, b10=0.75
         , b20=-0.6, b30=-0.8                                             \$
EOF
#=========================================> MAKE the ZEUS executable.
make -f makezeus
```

Note that a `#` in column 1 indicates a comment in a script file. In this example, two flavours of comment lines are used. Comments led with a double dashed line (`=======>`) indicate portions of the script file which rarely, if ever, need to be changed by the user. Comments with a single dashed line (`------->`) indicate portions of the script file that will probably need to be changed with every simulation. Below are descriptions of the six segments found in the script file `dzeus35.s`.

2.3.1 Files retrieved from the home directory

The first segment retrieves the files necessary to create the *ZEUS-3D* executable and are retrieved only if they do not already exist on disc [`if (! -e` *filename*`)`]. This example assumes that the script file is launched from the directory `zeus` created when `dzeus35.tar` from `www.ica.smu.ca/zeus3d` is unpacked. Files already in this directory and thus not

retrieved include:

`dzeus35`	the more than 90,000 lines of source code divided up into more than 350 subroutines
`zeus35.mac`	file containing all the *EDITOR* macros (§2.2)
`checkin.o`	the object file of the *C*-routine `checkin.c` (the only *C*-routine used by *ZEUS-3D*) which allows interrupt messages to be read from the terminal during interactive runs (§4)

while those retrieved from other directories include:

`xedit21`	the preprocessor executable
`dnamelist.a`	the double precision library of subroutines which emulate the *namelist* feature (§2.3.5)
`dsci01.a`	the double precision library of four specialised max-min subroutines
`grfx03.a`	library of subroutines calling routines in external graphics libraries (*NCAR* and *PSPLOT*)
`psplot.a`	library of routines for *PSPLOT* graphics
`nopsplot.a`	library of dummy *PSPLOT* routines used when the library `psplot.a` is not available or linked
`noncar.a`	library of dummy *NCAR* routines used when *NCAR* graphics are not installed at the site

2.3.2 Creating the `dzeus3.5` directory

The second segment creates the directory `dzeus3.5` on condition that it does not already exist. The precompiled source files (one subroutine per file) and the compiled object files are put here.

2.3.3 Creating the change deck `chgzeus`

The third segment creates the change deck `chgzeus` which is merged with the source code `dzeus35` during the preprocessing step. The first line in `chgzeus` reads the *EDITOR* macros in `zeus35.mac` using the *EDITOR* command `*read`. This command replaces the statement with the contents of the named file. Thus, the macros in `zeus35.mac` become part of the change deck `chgzeus`, and get merged with the source code. Next, the *EDITOR* command `*delete` (or `*d` for short) is used to replace lines 42 and 43 in the common deck `par` in the main source code `dzeus35` with the two following parameter statements which set the parameters to the desired values for the simulation. This is where the user should indicate the size of the arrays required for the simulation to be performed. The parameters set in the given example of the script file `dzeus35.s` are all described in §C.6.

Should the user have their own changes to the code, these can be most conveniently put into a file called `chguser`, for example, and the statement `**read chguser` would then be "de-commented" by deleting one of the asterisks. This will ensure that the user's changes will be incorporated just like those in `zeus35.mac` and the two parameter statements discussed above. Changes should be specified using the language of *EDITOR* (code prepared for

the old CTSS precompiler *HISTORIAN* can be processed by *EDITOR*), and would include additional subroutines such as the problem generator which need to be compiled with the rest of the source code. Full description of how to do this is found in §5.

2.3.4 Preprocessing `dzeus35`

The fourth segment creates the input deck for the preprocessor *EDITOR* and then fires it up. Changes to this segment should be needed rarely. If it becomes necessary to change the name of the main source file from `dzeus35`, or to change the name of the change deck from `chgzeus`, or to change the name of the directory created for the precompiled and compiled subroutine files from `dzeus3.5`, or to change the name of the makefile from `makezeus`, or to change the name of the *ZEUS-3D* executable from `xdzeus35`, or to use a compiler and loader other than the defaults (`f77` under *SUNOS*), these changes should be made in the *EDITOR* input deck `inedit`. In addition, various compiler options can be set as necessary. For example, the commented out (a "`c`" in column 1) `coptions` and `loptions` would allow full debugging under *SUNOS*, while the exposed (no "`c`" in column 1) `coptions` and `loptions` represent full optimisation for the *SUNOS* compiler. Note that lines "commented out" in a namelist will be echoed on the CRT as the input deck is read. This is a feature of the *EDITOR* `namelist`. (See §2.3.5 and App. B for a discussion of the *EDITOR* `namelist` feature.)

One last note on setting compiler options. On occasion, a few subroutines can cause a run to generate significantly different results when compiled with full optimisation than with little or no optimisation [often traceable to exponentiation (`**`)]. Examples of such "troublesome" routines in the code known to have this property include `corona`, `phistv`, and `couette`; there may well be others. In other cases, the time the compiler takes to optimise a particular routine may far exceed any run-time benefit. Examples of such routines include `plot1d`, `plot2d`, and `nmlsts`. A relatively new feature of *EDITOR* allows one to specify troublesome routines in `specdk` (a 1-D character*8 array) and the special compiler options to be used for these routines in `speccopt`. Thus, "de-commenting" the line:

```
c         , speccopt='-O1', specdk='corona','phistv','nmlsts','plot1d'
```

in the sample file `dzeus35.s` above would apply lesser optimistation (`-O1`) to the four routines listed, and full optimisation (`-O4`) to the rest. For additional details, the reader is referred to the `EDITOR` user manual: `edit21_man.ps`.

For parallel processing, set `iutask` (third line of the namelist `editpar`) to 1 for Cray *microtasking*, or 2 for *openMP*. This will cause *EDITOR* to insert the appropriate *scoping* commands at the beginning of the major do-loops in the code. One then has to set the appropriate compiler options for your compiler to compile the code for multiple processors.

Libraries are specified by setting `libs`, of which three examples are given in the sample file `dzeus35.s` above. The first and uncommented setting of `libs` requires only libraries included in `dzeus35.tar` (from `www.ica.smu.ca/zeus3d`); no systems or third-party libraries such as *NCAR* (for graphics, *e.g.*, §3.2, §3.3) or *HDF* (§3.6) are required. By virtue of the *PSPLOT* library, publication-quality and full-colour graphics are possible even without *NCAR* which, until recently, was the only graphics capability *ZEUS-3D* possessed. The second `libs` command (commented out) is the variation used at the ICA for *HDF* libraries,

while the third (also commented out) is when both *HDF* and *NCAR* are linked. Additional libraries may be linked by appending them to whatever `libs` list is used.

With this input deck, the preprocessor will merge the change deck `chgzeus` with `dzeus35`, carry out the precompiler commands according to the aliases and definitions in the macro file `zeus35.mac`, split up the precompiled source code (now containing nothing but *FORTRAN* syntax) into separate files for each subroutine, search the directory `dzeus3.5` and write to disc only those files which do not already exist or have been changed, and finally create the makefile `makezeus`, described in §2.3.6.

2.3.5 Creating the input deck `inzeus`

The fifth segment is where the input deck for the *ZEUS-3D* executable is created (`inzeus`) and so the user should set all input parameters here (described fully in App. B). In this example, `inzeus` is set up for the 1-D MHD Brio and Wu shock tube problem. *ZEUS-3D* uses namelists to specify input parameters but does not use the standard `namelist` utility. Historically, the first versions of `namelist` available under *UNICOS* were horrid (character variables could not be set, vectors could only be set one element at a time, error messages were unreadable), and so a more useful `namelist` utility was incorporated into the preprocessor *EDITOR*. Thus, as one of its duties, *EDITOR* can be instructed (`inmlst=1`) to replace all references to namelists with calls to subroutines found in the library `dnamelist.a` which is linked to the executable during the *MAKE* process. This step is entirely transparent to the user. Namelists can be used as always, with the usual (more or less) syntax, bearing in mind that once defined, a namelist must be read before the next namelist is defined. Since this time, `namelist` has become a standard feature of *FORTRAN90* and has been significantly improved. Should the user prefer to use the `namelist` utility of the local OS, then the input parameter `inmlst` in the *EDITOR* input deck `inedit` should be set to 0 (§2.3.4). Be warned that doing this may make some of the namelists in the `dzeus35.s` (`inzeus`) file unreadable and generate run-time error messages. Syntactic errors may even arise during compilation.

One major difference between the *FORTRAN90* `namelist` and the *EDITOR* `namelist` is the latter allows for rank 2 arrays to be specified in an extremely intuitive fashion. For example, to set `((diib1(i,j),i=20,30),j=70,80)` to 1.0, while setting the rest of the 100 by 100 array to 0.1, one merely needs to type:

```
diib1(1:100,1:100)=0.1, diib1(20:30,70:80)=1.0
```

where the order is important. This capacity is not supported by *FORTRAN90*, and so some of the `namelist` syntax will have to be changed in the input decks `inzeus` and `inedit` should the user wish to use the standard `namelist`. If using the `EDITOR namelist` feature, remember not to allow any of the namelist lines to extend beyond the 72nd column. The first column in each line can be a blank or a 'c' (to comment out the line) and nothing else. The second column may contain a blank or a '$' and nothing else. (Note that because `dzeus35.s` is a script file, the $ must be "protected" by a \. Otherwise, the script file will try to interpret the $ as a control character rather than treating it as a character to be written to a disc file. The user will note that a \ does not precede the $ in the input deck `inzeus` once it is written to disc by `dzeus35.s`.) Text specifying the input parameters may start in column 3. If a character string is too long to fit in the 72 column format, one simply

types as much as one can in the first line (*i.e.*, up to and including the 72nd column), then resumes typing the character string on the next line, beginning in column 3. A single quote must appear before the first character in the first line of the character string and after the last character in the last line of the character string only.

A detailed description of all the namelist parameters is contained in App. B.

2.3.6 Making the executable `xdzeus35`

The sixth and final segment fires up the makefile `makezeus` created by the preprocessor *EDITOR*. The makefile will compile only those *FORTRAN* files in the directory `dzeus3.5` which have been written since the last time they were compiled, then link all the object files together with the specified libraries to create the executable `xdzeus35`.

2.4 Executing *ZEUS-3D*

Once the script file has completed successfully, simply type `xdzeus35` followed by a carriage return, and *ZEUS-3D* will begin running. In general, one can move the two files `xdzeus35` and `inzeus` to any other directory and the executable can be launched from that directory simply by typing `xdzeus35`, followed by a carriage return (enter).

Alternatively, one can run *ZEUS-3D* in batch mode, and for this the user should consult their systems administrator as batch facilities are highly system and installation dependent.

3 Output from *ZEUS-3D*

A variety of methods for dumping data to disc during execution are available in *ZEUS-3D*. Each of these methods has their specific use, and at times all types are used simultaneously. In this section, a brief description of each method is given, along with a list of the most vital statistics. These include: the *EDITOR* definition (if any) which enables the data dump, the logical unit to which the dumps are attached during execution, the namelist which controls the data dump (App. B), the convention used for naming the disc file for this type of data dump, and the format of the data in the disc file created.

3.1 Restart dumps

These are full precision dumps of all variables at specified time intervals which can be used to resume a calculation should a job terminate prematurely for whatever reason. Note that for runs where the total energy density equation is being solved (`itote=1`), only the internal energy density is actually stored since the total energy density is easily recomputed from the primitive variables. Similarly, even if the vector potential variation of the induction equation is being used (`VECPOT` defined), only the magnetic field components are stored.

Execution can be instructed to overwrite the previous even (odd) numbered dump with the new even (odd) numbered dump should disc space be limited. Thus, only two restart dumps would exist at any one time. Anticipate the size of a restart dumps to be about $10 \times$ `in` $\times$ `jn` $\times$ `kn` words for MHD runs and $6.5 \times$ `in` $\times$ `jn` $\times$ `kn` words for HD runs.

The first data written to a restart dump are the array dimensions and parameters which indicate which *EDITOR* macros are defined. Values of *EDITOR* aliases are not stored. These, then, are the first data read from a restart dump and are used to allow a restart dump to be read regardless of the differences between the array dimensions and *EDITOR* definition settings in the new executable (that which is reading the restart dump) and the old executable (that which created the restart dump). Thus, it is possible, for example, to resume an MHD run without the `MHD` definition set (and thus resume the calculation hydrodynamically), or to read the inner eighth of a 64^3 data volume into any part of a new 128^3 grid, or whatever.

EDITOR definition:	none
logical unit:	`iodmp`
namelist:	`rescon`
filename:	`zr`*nnn*`id`, where `zr` is the common prefix to all restart dumps, *nnn* is a three digit integer distinguishing the multiple dumps created during a run, and `id` is a two character, user-specified problem tag.
format:	binary, one word (8 bytes) per datum

3.2 1-D plot files

These are metacode (*NCAR*) or postscript (*PSPLOT*) files each of which contains publication-quality 1-D plots along one of the specified 1-D slices through all of the selected variables. If, for example, m slices are specified for n variables, then each time 1-D plots are

required, m files will be created each containing n plots.

EDITOR definition:	`PLT1D`
logical unit:	`ioplt1`
namelist:	`plt1con`
filename:	`zp`*nnn*`id`.*mm*, where `zp` is the common prefix to all 1-D plot files, *nnn* and `id` are as defined for restart dumps, and *mm* is an extension indicating the slice number. For *PSPLOT*, the suffix `.ps` is added to the filename.
format:	metacode—use `idt` to read *NCAR*-generated metafiles postscript—use `mgv/ggv` to read *PSPLOT*-generated postscript files

3.3 2-D plot files

These are metacode (*NCAR*) or postscript (*PSPLOT*) files each of which contains publication-quality 2-D plots (contours and/or vectors) on one of the specified 2-D slices through all of the selected variables. If, for example, m slices are specified for n variables, then each time 2-D plots are required, m files will be created each containing n plots.

EDITOR definition:	`PLT2D`
logical unit:	`ioplt2`
namelist:	`plt2con`
filename:	`zq`*nnn*`id`.*mm*, where `zq` is the common prefix to all 2-D plot files, *nnn* and `id` are as defined for restart dumps, and *mm* is an extension indicating the slice number. For *PSPLOT*, the suffix `.ps` is added to the filename.
format:	metacode—use `idt` to read *NCAR*-generated metafiles postscript—use `mgv/ggv` to read *PSPLOT*-generated postscript files

3.4 2-D pixel dumps

These are "binned" 2-D slices through the data volume of a single variable designed for visualisation. They can be written in either raw format (one byte per datum) or *HDF* (four bytes per datum). The raw format files can be read by *XImage* and are not intended for quantitative analysis since the dynamic range (256) is too small for most purposes other than qualitative rendering. The *HDF* files may be read by *XImage* as well, or any other software package capable of reading *HDF* files and may be used quantitatively. Polar plots are rebinned to a Cartesian plane, and dumped as Cartesian pixel plots. Because the data files are so small (especially the raw format), enough images can be written to disc during the simulation to create a smooth temporal animation of the calculation for a number of variables. Multiple slices can be specified for each variable and, in a post-processing session using *DATAVU* (a program available from the author which formats and annotates frames for an animation), reassembled in their proper 3-D perspective. Note that raw pixel dumps have no header. Thus, the dimensions of the dumps (needed to read the raw dumps correctly)

are noted in the message log file (see below) each time a dump is created.

EDITOR definition:	`PIX`
logical unit:	`iopix`
namelist:	`pixcon`
filename:	`zi**`*nnn*`id.`*mm*`.h`, where `zi` is the common prefix to all 2-D pixel dumps, `**` is a two-character representation of the variable (see Table 3.1 in §3.12), *nnn* and `id` are as defined for restart dumps, *mm* is an extension indicating the slice number, and `h` is an extension added *only* for *HDF* files.
formats:	raw (one byte per datum); or *HDF* (four bytes per datum)

3.5 3-D voxel dumps

These are 3-D dumps of a single variable rebinned to a Cartesian grid using either raw format (one byte per datum) or *HDF* (four bytes per datum). These are the 3-D analogues of the 2-D pixel dumps and can be used by a variety of software packages including *DATAVU* and Spyglass *DICER*. In this release, voxel dumps may be generated in both Cartesian (`XYZ`) and cylindrical (`ZRP`) coordinates. Storing enough of these images to create a smooth 3-D animation of a run is possible, but may strain local disc space limitations. As much as 4 Megabytes per raw-format image may be required for a one million zone simulation. Note that the maximum dimensions of a voxel dump are `in-1, 2*jn-1, 2*kn-1`. Since raw voxel dumps have no header, software reading these dumps will require their dimensions as input. These are noted in the message log file as the voxel dumps are created.

EDITOR definition:	`VOX`
logical unit:	`iovox`
namelist:	`voxcon`
filename:	`zv**`*nnn*`id.h`, where `zv` is the common prefix to all 3-D voxel dumps, `**`, *nnn*, `id`, and `h` are as defined for pixel dumps.
formats:	raw (one byte per datum); or *HDF* (four bytes per datum)

3.6 HDF files

HDF (Hierarchical Data Format) files contain 3-D data of one or more variables in the *HDF* format developed at the NCSA, and differs from the voxel *HDF* dumps in that these dumps are not rebinned. The data are stored in four byte words which is more than adequate for quantitative graphical study. Most graphical software packages at the NCSA use this format for data dumps. *HDF* files are useful because they contain header information which include array dimensions, extrema of data, and the grid coordinates. The size of an *HDF* file containing a single variable is the number of active zones times 4 bytes. For a "total" dump (all primary variables to the same *HDF* file) with none of `GRAV`, `PSGRAV`, or `TWOFLUID` defined, the size is the number of active zones times 32 bytes for MHD runs, or times 20 bytes for HD runs.

EDITOR definition:	`HDF`
logical unit:	none
namelist:	`hdfcon`
filename:	`zh**`*nnn*`id`, where `zh` is the common prefix to all *HDF* files, `**`, *nnn*, and `id` are as defined for pixel dumps.
format:	*HDF*, four bytes per datum

3.7 Time slice dumpfiles

There are two types of time slice dumps, and either, both, or neither may be selected. The first is a single ascii file which contains values of various scalars at specified time intervals. The second is a file (metacode or postscript) containing 1-D plots of these scalars plotted as a function of time. The user selects the time interval for the ascii and plot dumps independently. The scalars include various integral quantities such as total mass, angular momenta, magnetic monopoles, energy, *etc.*, as well as extrema of quantities such as density, pressure, divergence of magnetic field, *etc.* The user may wish to add other scalars to this format (subroutines `tslice` and `tslplot`).

EDITOR definition:	`TIMESL`
logical units:	`iotsl` and `iotslp`
namelist:	`tslcon`
filenames:	`zt`*ll*`id` (ascii file), where `zt` is the common prefix to all time slice ascii files, *ll* is incremented by one each time the job is restarted, and `id` is as defined for restart dumps. `ztp`*ll*`id` (plot file), where `ztp` is the common prefix to all time slice plots.
formats:	ascii and metacode/postscript

3.8 Display dumpfiles

Display dumps are single ascii files (maximum of 80 characters per line) which contains a quantitative display (matrix format) of a specified portion of various 2-D slices through any of many variables at evenly spaced time slices during a simulation. The data are scaled and converted to integers before being written to the ascii file. The dynamic range of the scaled data depends on the specified "width" of the field of view (no more than 38), and ranges from 10^2 to 10^6. For very small widths (≤ 8), the data are not scaled and written as real numbers, with three or four significant figures. This utility is much like `PRTIM` in *AIPS*, for those familiar with the Astronomical Image Processing System. Its primary use is in debugging, or when one needs to view a small portion of data quantitatively and simultaneously.

EDITOR definition:	`DISP`
logical unit:	`iodis`
namelist:	`discon`
filename:	`zd`*ll*`id`, where `zd` is the common prefix to all display files, *ll* is as defined for time slice dumps, and `id` is as defined for restart dumps.
format:	ascii

3.9 *RADIO* dumps

RADIO[3] dumps are similar to the 2-D pixel dumps, but contain line-of-sight integrations of various quantities rather than 2-D slices through the data volume. In this release, *RADIO* dumps are possible in both Cartesian (`XYZ`) and cylindrical (`ZRP`) coordinates (though the latter are not fully debugged). The integrands are all scalars (bremsstrahlung, density, internal energy, magnetic pressure, specific internal energy, velocity shear, velocity divergence, and three Stokes emissivities) and are integrated using a very fast binning algorithm that is as much as 50 times faster than traditional direct ray-tracing algorithms. Files may be dumped in either raw format (one byte per datum) or *HDF* (four bytes per datum).

EDITOR definition:	`RADIO`
logical unit:	`iorad`
namelist:	`radcon`
filename:	`zR**`*nnn*`id.h`, where `zR` is the common prefix to all *RADIO* dumps, `**`, *nnn*, `id`, and `h` are as defined for pixel dumps.
formats:	raw (one byte per datum); or *HDF* (four bytes per datum)

3.10 Message log files

The message log file contains all the messages that are written to the terminal by the code during execution. In addition, the grid and all the values of the namelist parameters specified in the file `inzeus` are dumped here. It serves as the log for the execution.

EDITOR definition:	none
logical unit:	`iolog`
namelist:	`none`
filename:	`zl`*ll*`id`, where `zl` is the common prefix to all log files, *ll* is as defined for time slice dumps, and `id` is as defined for restart dumps.
format:	ascii

3.11 Userdump

`USERDUMP` is an *EDITOR* alias available for the user to include their own special type of I/O which may be desired in addition to those currently available. See §5 for details on how to add subroutines to the code.

EDITOR definition:	none
logical unit:	`iousr`
namelist:	`usrcon`
filename:	`zu`*nnn*`id`, where `zu` is the common prefix to all user dump files, *nnn* and `id` are as defined for restart dumps.
format:	

[3]The original post-processing program, *RADIO*, was designed to take line-of-sight integrations through an MHD datacube to compute the Stokes parameters, and thus mimic *radio* observations from telescopes such as the VLA, whence the name.

3.12 Recognised plotting variables

Table 3.1 below and continued on the following page lists the two-character variable representations [corresponding to the double asterisks (**) used in §3.4, §3.5, §3.6, and §3.9 above] used for generating the filenames for pixel (P), voxel (V), *HDF* (H), and *RADIO* (R) dumps. These two-character representations are identical to those used to specify the variables to be dumped (see `pixvar` in namelist `pixcon`, `voxvar` in namelist `voxcon`, `hdfvar` in namelist `hdfcon`, and `radvar` in namelist `radcon`, Appendix 2) with the exception that variables specified by a single character (*e.g.*, `d`) appear with a trailing underscore (*e.g.*, `d_`) in the dump file name. The third column indicates the I/O types in which the variable may be dumped.

Table 3.1 Two Character Variable Representations

**	Variable	Dumps	**	Variable	Dumps
a_	vector potential norm	PVH	m_	Mach number	PVH
a1	1-vector potential	PVH	ma	Alfvénic Mach number	PVH
a2	2-vector potential	PVH	mf	fast magnetosonic number	PVH
a3	3-vector potential	PVH	ms	slow magnetosonic number	PVH
an	normal vector pot.	P	p1	1st thermal pressure	PVH
ap	poloidal vector pot.	P	p2	magnetic pressure	PVH
b_	magnetic field norm	PVH	p3	1st thermal + magnetic pres.	PVH
b1	1-magnetic field	PVH	p4	2nd thermal pressure	PVH
b2	2-magnetic field	PVH	p5	1st + 2nd thermal pressures	PVH
b3	3-magnetic field	PVH	p6	2nd thermal + magnetic pres.	PVH
bP	ϕ-magnetic field	P	p7	1st + 2nd + magnetic pres.	PVH
bR	radial magnetic field	P	pa	pitch angle; $\tan^{-1}(B_1/B_\phi)$	P
bn	normal magnetic field	P	pg	pseudo-grav. potential	PVH
bp	poloidal magnetic field	P	s1	1-momentum	PVH
bt	plasma beta = $2p/B^2$	PVH	s2	2-momentum	PVH
d_	density	PVH	s3	3-momentum	PVH
e1	first internal energy	PVH	sd	skew-density	P
e2	second internal energy	PVH	sn	normal momentum	P
et	total energy density	PVH	sp	poloidal momentum	P
fn	normal flux function	P	to	all field arrays	H
gp	gravitational potential	PVH	u1	1st specific int. energy	PVH
j_	current density norm	PVH	u2	2nd specific int. energy	PVH
j1	1-current density	PVH	v_	velocity norm (speed)	PVH
j2	2-current density	PVH	v1	1-velocity	PVH
j3	3-current density	PVH	v2	2-velocity	PVH
jn	normal current density	P	v3	3-velocity	PVH
jp	poloidal current density	P	vn	normal velocity	P
k1	first specific entropy	PVH	vp	poloidal velocity	P
k2	second specific entropy	PVH	vv	velocity divergence	PVH
ka	averaged specific entropy	PVH	w	vorticity norm	PVH

Table 3.1, continued. Two Character Variable Representations

**	Variable	Dumps	**	Variable	Dumps
w1	1-vorticity	PVH	IV	I with pol'n vectors	R
w2	2-vorticity	PVH	M_	Mach Number	R
w3	3-vorticity	PVH	MA	Alfvénic Mach number	R
wn	normal vorticity	P	MF	fast magnetosonic Number	R
wp	poloidal vorticity	P	MS	slow magnetosonic Number	R
A_	pol'n position angle	R	P_	polarised intensity	R
AV	A with pol'n vectors	R	PV	P with pol'n vectors	R
B_	magnetic field norm	R	SH	scalar velocity shear	R
BR	bremsstrahlung	R	U1	1st sp. int. energy (temp.)	R
D_	density	R	V_	pol'n vectors (black)	R
E1	1st internal energy (pres.)	R	VR	pol'n vectors (white)	R
F_	fractional pol'n	R	VV	velocity divergence	R
FV	F with pol'n vectors	R	W_	vorticity norm	R
I_	total intensity	R			

4 Interacting with *ZEUS-3D*

During an interactive execution (as opposed to batch), the user may probe *ZEUS-3D* for its status, change input parameters, and submit instructions to create a dump, stop, pause, resume, *etc.* This is done by typing a recognised three-character "interrupt message" followed by a carriage return. Once every "time step", *ZEUS-3D* "glances" at the terminal buffer (by virtue of the lone *C* routine `checkin.c` introduced in §2.3.1). If an interrupt message has been entered, *ZEUS-3D* will carry out the instruction. If no interrupt message is found, execution proceeds without pause. Below is a list of the interrupt messages recognised by *ZEUS-3D*, along with a brief description of their function. Only the first three characters of each command (those in `typewriter` font) need be entered. Note that there are several synonyms for a number of the commands, which are separated by commas.

Controlling execution:

- `time`, `cycle`, `status`, `t`, `n`, `?`
  prints a time and cycle report, then resumes execution
- `quit`, `abort`, `crash`, `break`
  immediate emergency termination, no final dumps are made
- `stop`, `end`, `exit`, `finish`, `terminate`
  clean stop—all final dumps are made
- `halt`, `pause`, `wait`, `interrupt`
  halt execution and wait for a message from the crt or controller.
- `restart`, `go`
  restarts execution after a halt
- `tlimit`, `tfinish` (followed by a real number)
  resets the physical (problem) time limit (when computation will stop)
- `nlimit`, `nfinish` (followed by an integer)
  resets the cycle limit
- `ttotal`, `tcpu` (followed by an integer number of seconds)
  resets maximum cpu time to consume.
- `tsave`, `treserve` (followed by an integer number of seconds)
  resets the save time reserved for cleanup and termination

Controlling data output:

- `dump`
  creates a restart dump at current time
- `dtdmp` (followed by a real time interval)
  resets the problem time interval between restart dumps

- `pl1`
  creates a 1-D plot at current time
- `dt1` (followed by a real time interval)
  resets the problem time interval between 1-D plots
- `pl2`
  creates a 2-D plot at current time
- `dt2` (followed by a real time interval)
  resets the problem time interval between 2-D plots
- `pix`el
  creates a pixel dump at current time
- `dtpix` (followed by a real time interval)
  resets the problem time between pixel dumps
- `vox`el
  creates a voxel dump at current time
- `dtvox` (followed by a real time interval)
  resets the problem time between voxel dumps
- `usr`
  creates a user dump (calls `USERDUMP`) at current time
- `dtusr` (followed by a real time interval)
  resets the problem time between user dumps
- `hdf`
  creates an *HDF* dump at current time
- `dth` (followed by a real time interval)
  resets the problem time between *HDF* dumps
- `tsl`ice
  adds a time slice dump at current time to time slice file
- `dtts`lice (followed by a real time interval)
  > 0 $\Rightarrow$ resets the problem time between time slice ascii dumps
  < 0 $\Rightarrow$ resets the problem time between time slice plot dumps
- `dis`play
  adds a display dump at current time to display dump file
- `dtd`isplay (followed by a real time interval)
  resets the problem time between display dumps
- `rad`io
  creates a radio dump at current time
- `dtr`adio (followed by a real time interval)
  resets the problem time between radio dumps

5 Adding source code to *ZEUS-3D*

5.1 Adding an entire subroutine

Adding source code to the *ZEUS-3D* package is not as difficult as one might think, especially if all one wants to do is add new subroutines or replace existing ones. Below is the subroutine `myprob` which can be used as a template to create a problem generator. A soft copy of `myprob` may be found in the `zeus` directory of `dzeus35.tar` from `www.ica.smu.ca/zeus3d`. The style is that which is used for all subroutines currently in `dzeus35`.

```
*insert zeus3d.9999
*deck myprob
c=======================================================================
c
c    \\\\\\\\\\      B E G I N   S U B R O U T I N E      //////////
c    //////////                M Y P R O B                \\\\\\\\\\
c
c=======================================================================
c
       subroutine myprob
c
c    abcd:zeus3d.myprob <------------------------ initialises my problem
c                                                        september, 1990
c
c    written by: A Busy Code Developer
c    modified 1: December 1993, by ABCD, modified for two fluids
c    modified 2: August 2007, by ABCD, modified for new magnetic
c                boundary conditions
c
c  PURPOSE:  Initialises all the flow variables for my problem.  More
c  description of my problem can go here.
c
c  LOCAL VARIABLES:
c
c-----------------------------------------------------------------------
c
*call comvar
       integer       i       , j       , k
       real          da      , db      , e1a     , e1b     , e2a
     1             , e2b     , v1a     , v1b     , v2a     , v2b
     2             , v3a     , b1a     , b1b     , b2a     , v3b
     3             , b2b     , b3a     , b3b
*if def,MHD
     4             , q11     , q12     , q2      , q3
*endif MHD
c
c      The following arrays are never used, and are placed here only to
c  show how arrays can be declared and then equivalenced to "global
c  worker arrays" so that the size of the executable is not increased.
c
       real          array1d (ijkx)
       real          array2d (idim,jdim)
       real          array3d (  in,  jn,  kn)
c
       equivalence   ( array1d , wa1d     )
       equivalence   ( array2d , wa2d     )
       equivalence   ( array3d , wa3d     )
```

```
c
c-----------------------------------------------------------------------
c
c  Input parameters:
c
c    da , db    array and boundary values for density
c    e1a, e1b   array and boundary values for first  internal energy
c    e2a, e2b   array and boundary values for second internal energy
c    v1a, v1b   array and boundary values for 1-velocity
c    v2a, v2b   array and boundary values for 2-velocity
c    v3a, v3b   array and boundary values for 3-velocity
c    b1a, b1b   array and boundary values for 1-magnetic field
c    b2a, b2b   array and boundary values for 2-magnetic field
c    b3a, b3b   array and boundary values for 3-magnetic field
c
      namelist / pgen     /
     1               da      , db      , e1a     , e1b     , e2a
     2             , e2b     , v1a     , v1b     , v2a     , v2b
     3             , v3a     , b1a     , b1b     , b2a     , b2b
     4             , b3a     , b3b
c
c     Set default values
c
      da  = 1.0
      db  = 0.1
      e1a = 0.9
      e1b = 9.0
      e2a = 0.0
      e2b = 0.0
      v1a = 0.0
      v1b = 1.0
      v2a = 0.0
      v2b = 1.0
      v3a = 0.0
      v3b = 1.0
      b1a = 0.0
      b1b = 0.0
      b2a = 0.0
      b2b = 0.0
      b3a = 0.0
      b3b = 0.0
c
c     Read namelist pgen.
c
      read  (ioin , pgen)
      write (iolog, pgen)
c
c     Set field arrays.  Metric factors in the magnetic field settings
c  are necessary to preserve the solenoidal condition.  Note that the
c  first internal energy is initialised even if the total energy
c  equation is being solved.  If needed, routine TOTNRG is called by
c  SETUP to initialise the total energy "et".
c
      do 30 k=ksmnm2,kemxp3
        do 20 j=jsmnm2,jemxp3
          do 10 i=ismnm2,iemxp3
            d (i,j,k) = da
            v1(i,j,k) = v1a
            v2(i,j,k) = v2a
```

```
              v3(i,j,k) = v3a
*if -def,ISO
              e1(i,j,k) = e1a
*endif -ISO
*if def,TWOFLUID
              e2(i,j,k) = e2a
*endif TWOFLUID
*if def,MHD
              b1(i,j,k) = b1a
              b2(i,j,k) = b2a * g2bi (i)
              b3(i,j,k) = b3a * g31bi(i) * g32bi(j)
*endif MHD
10          continue
20        continue
30      continue
*if -def,ISYM
c
c       Set inflow boundary arrays.
c
*if def,MHD
        q11 = ( v2b * b3b - v3b * b2b ) * dx1a(ism1)
        q12 = ( v2b * b3b - v3b * b2b ) * dx1a(ism2)
        q2  = ( v3b * b1b - v1b * b3b ) * g2a (is  )
        q3  = ( v1b * b2b - v2b * b1b ) * g31a(is  )
*endif MHD
        do 50 k=ksmnm2,kemxp3
          do 40 j=jsmnm2,jemxp3
            niib    (j,k) = 3
            diib1   (j,k) = db
            diib2   (j,k) = db
            v1iib1  (j,k) = v1b
            v1iib2  (j,k) = v1b
            v1iib3  (j,k) = v1b
            v2iib1  (j,k) = v2b
            v2iib2  (j,k) = v2b
            v3iib1  (j,k) = v2b
            v3iib2  (j,k) = v2b
*if -def,ISO
            e1iib1  (j,k) = e1b
            e1iib2  (j,k) = e1b
*endif -ISO
*if def,TWOFLUID
            e2iib1  (j,k) = e2b
            e2iib2  (j,k) = e2b
*endif TWOFLUID
*if def,MHD
            b2iib1  (j,k) = b2b
            b2iib2  (j,k) = b2b
            b3iib1  (j,k) = b3b
            b3iib2  (j,k) = b3b
            emf1iib1(j,k) = q11
            emf1iib2(j,k) = q12
            emf2iib1(j,k) = q2 * dx2a(j)
            emf3iib1(j,k) = q3 * dx3a(k) * g32a(j)
*endif MHD
40        continue
50      continue
*endif -ISYM
c
```

```
       write (iotty, 2010)
       write (iolog, 2010)
2010   format('MYPROB  : Initialisation complete.')
c
       return
       end
c
c=======================================================================
c
c    \\\\\\\\\\        E N D   S U B R O U T I N E        //////////
c    //////////                M Y P R O B                \\\\\\\\\\
c
c=======================================================================
c
c
```

There are many ingredients to this template which warrant discussion. In order of appearance, these are:

1. Ignoring for the moment the *EDITOR* statement `*insert zeus3d.9999`, the first line of each subroutine must be an *EDITOR* `*deck` (`*dk` for short) statement. Without this statement, the precompiler won't put the subroutine into a separate file, inhibiting the debugger should it be necessary. It is easiest, although not necessary, to give the deck the same name as the subroutine.

2. Note that there is no parameter list in the subroutine statement. A parameter list is unnecessary since all variables that need to be used and/or set are accessible via the common blocks. In fact, using a parameter list would inhibit the inclusion of a user-supplied subroutine using the present structure of the code.

3. All of the important variables declared in `dzeus35` are in common blocks, and can be included into a subroutine simply by inserting the *EDITOR* statement `*call comvar` just before the local declarations are made. The *EDITOR* `*call` (`*ca` for short) statement is much like `INCLUDE` whereby a section of code known as a "common deck" (called `comvar` in this case) is inserted at the location of the `*call` statement. Every variable of any possible interest is declared in `comvar`, including many that the user would never need. (A description of the most widely used variables is given in App. C.) At the beginning of `comvar` is an "`implicit none`" statement, which requires that the attributes of all variables used in the subroutine be declared. Note that should the user inadvertently try to use a variable name already declared in `comvar`, the compiler will flag the repetition and abort compilation. While the "`implicit none`" does not require that all externals called by the program unit be declared in an `external` statement, it is still good practise to do so. In fact, if undeclared externals appear inside a nested do-loop construct, this may inhibit *EDITOR*'s auto-tasking feature (parameter `iutask`; see §2.3).

4. Should one dimensional arrays be required to store data at each grid point along one of the axes, it is best to declare the 1-D vector with dimension `(ijkx)`, as done in the template. The parameter `ijkx` is declared in `comvar` and is defined as the largest of

`in`, `jn`, and `kn` (the dimensions of the 3-D arrays), also declared in `comvar`. So that no additional memory is occupied by this local array, it can be equivalenced to one of the 26 1-D scratch arrays declared in `comvar`, as done in the template. The names of all the scratch arrays (1-D, 2-D, and 3-D) are given in §C.4 and their dimensions (*e.g.*, `idim` and `jdim`) are defined in §C.6.

5. The namelist `pgen` is reserved for the namelist in the Problem `GEN`erator. Of course, any name other than `pgen` could be used, so long as it is not already used in the input deck `inzeus` and the new name for the namelist is substituted for `pgen` in `inzeus`. Note how default values for the input parameters can be assigned before the namelist is read.

6. Loop 30 is a typical way the 3-D field variables (`d` = density, `e1` = first internal energy per unit volume, *etc.*) are assigned values. In this very simple case, the variables are assigned to the scalars read from the namelist `pgen`. Note that all energy variables (*e.g.*, `e1`, `e1iib1`, *etc.*) should be considered as energy per unit *volume* and not energy per unit *mass*. Appendix C has a list of all the variable names and their dimensions. The do-loop indices declared in `comvar` are all assigned values in the subroutine `nmlsts` which is called immediately before ther user's problem generator (`PROBLEM`) is called (see App. A) and so they can be used explicitly in any user-supplied subroutine called thereafter. Thus, the index for loop 30 (`k`) ranges from `ksmnm2` (k-start minimum minus 2) to `kemxp3` (k-end maximum plus 3), which includes all boundary zones. This is particularly important for the magnetic field variables. Similarly for the indices of loops 20 (`j`) and 10 (`i`). Note the use of the *EDITOR* `*if define`, `*endif` (`*if def`, `*ei` for short) structure which conditionally includes or excludes a segment of coding depending on whether, in this case, `MHD` was defined during precompilation. Similar conditionals can be based on the "truth" of any *EDITOR* definition, and on how aliases are set. For example, one could place an *EDITOR* `*if alias PROBLEM.eq.myprob` just after the `subroutine` statement, and the matching `*endif` just before the `return` statement. In this way, the subroutine would be empty (nothing between the `subroutine` and `return` statements) unless the *EDITOR* alias `PROBLEM` were set to `myprob`. This would prevent it from being compiled when it is not needed.

7. Loop 50 illustrates how inflow boundary values (to be applied only to those boundary zones where matter is flowing into the grid in a known fashion) can be set for super-magnetosonic flow. (See §§1.5 and B.8 for variations required for submagnetosonic inflow conditions.) In this case, the "inner-i-boundary" (`iib`) values of the flow variables are being initialised. Alternatively, one could set the in-flow boundary values as input parameters using the namelists `iib`, `oib`, *etc.* (§B.8, §B.9, *etc.*). Note the use of the *EDITOR* `*if define`, `*endif` construct to prevent this loop from being compiled in the event that `ISYM` is defined. If `ISYM` has been defined, the variables `niib`, *etc.* are *not* declared in `comvar`. Variables that are conditionally declared (depending on which *EDITOR* definitions are set) are noted in App. C.

8. Finally, if desired, the user can write various messages to the terminal (logical unit `iotty`) or to the message log file (logical unit `iolog`). Both `iotty` and `iolog` are

declared in `comvar` and set by the subroutine `mstart`, and thus available in `PROBLEM` so long as this subroutine starts off with `*ca comvar` as exemplified in `myprob`.

9. **New to Version 3.5:** The routines `bndyflgs` and `bndyall` (which set all boundary values after the 3-D arrays have been set) are now called after `PROBLEM` in subroutine `SETUP`, and thus the user need not include these calls in their problem generator. Accordingly, calls to `bndyflgs` and `bndyall` have been omitted from the template `myprob` given above.

Once the subroutine is written, it should be placed in its entirety into a change deck called, for example, `chguser` and the line `**read chguser` in the script file `dzeus35.s` should be "de-commented" by deleting one of the asterisks (§2.3). Upon its first pass (the merge step), the preprocessor will, in this case, insert the user's subroutine into `dzeus35` immediately after line 9,999 of the main program `zeus3d` (by virtue of the *EDITOR* statement `*insert zeus3d.9999` appearing at the top of the subroutine template). Since `zeus3d` doesn't have 9,999 lines, *EDITOR* will simply stick the subroutine after the last line of the main program. It doesn't matter where in `dzeus35` the subroutine gets inserted so long as it isn't in the middle of an existing subroutine (deck). Immediately after the main program is as good a place as any. Upon the second pass, the precompiler will find the user's subroutines and treat them as it would any other it encounters. Thus, if there are any *EDITOR* commands in the user's routines (such as `*call comvar`, `*if define,MHD`), they will be carried out and then expunged from the working copy of the source code. The user's subroutine will then be placed in its own file in the directory `dzeus3.5`, and the name of the subroutine will be included in the makefile `makezeus` which will then compile the subroutine and link it with the rest of the object files and libraries. Provided the *EDITOR* alias `PROBLEM` has been set to `myprob` (or whatever it's called) in the macro file `zeus35.mac`, the user's problem generator will be called at the appropriate time during execution. Similarly, if the subroutine should be called at the location of any of the other available "plugs" in the code, set the appropriate alias (*i.e.* `SPECIAL`, `SPECIALSRC`, `USERSOURCE`, `SPECIALTRN`, `USERDUMP`, `PROBLEM`, `PROBLEMRESTART`, or `FINISH`; see §2.2.2 and the *ZEUS-3D* skeleton in App. A) in `zeus35.mac` to the subroutine name.

5.2 Microsurgery using *EDITOR*

For the truly adventurous, it is possible to alter individual lines of code in `dzeus35` without actually changing the original source code. In this way, the changes made can be kept separate from the code, and thus not lost in the abyss of `dzeus35`. In addition, the user's changes could, in principle, be incorporated into the master code at a later date and become part of the next release. To do this, there are two things required: an *EDITOR* listing of the code and a short tutorial on how to use *EDITOR*. For those who have worked with *HISTORIAN*, all this should seem very familiar. For those who haven't, take heart—the structure is very intuitive. The real problem will be ensuring that the changes made don't break something else in the code. This is where the headaches will lie, and those who really want to change the code do so at their own peril!

To get an *EDITOR* listing of the code, run the script file `number.s` (a soft-copy of which may be found in the `editor` directory of `dzeus35.tar` from `www.ica.smu.ca/zeus3d`):

```
#============ SOURCE FILE TO CREATE A NUMBERED LISTING ==============#
#
#=======================================> Get files from home directory.
if(! -e xedit21) cp ../editor/xedit21 .
#----------------------> Create the input deck for EDITOR, and execute.
rm -f inedit
cat << EOF > inedit
 \$editpar   inname='dzeus35'
          , ibanner=1, job=1, inumber=3, itable=1, ixclude=1          \$
EOF
chmod 755 xedit21
./xedit21
```

by typing:

```
csh -v number.s
```

This script file will fire up *EDITOR* in its numbering mode (`job=1`), and produce a listing with a table of contents, and various labels on each line. The numbered file will be called `dzeus35.n`, and can be viewed in a wide (132 character) window. Printed copy is not recommended; at 60 lines per page, there will be more than 1,500 pages of output! The third column to the right of the source listing is the number of lines since the most recent *EDITOR* `*deck` or `*cdeck` statement. This is the column needed to perform microsurgery on the master file.

During preprocessing, *EDITOR* makes two major passes over the code. The first pass does the merging of the change deck `chgzeus` (which contains `zeus35.mac` and possibly `chguser`) into the main code. *EDITOR* commands performed during this pass include:

1. `*insert` *deckname.n*—inserts text immediately following the `*insert` command into the source code directly after line n in deck (or cdeck: common deck) *deckname*. The value of n is determined from the third column to the right of the source code in the numbered listing, `dzeus35.n`.

2. `*delete` *deckname.n,m*—deletes lines n through m in deck (or cdeck) *deckname*, and replaces it with the text immediately following the `*delete` command, if any. Note that m must be greater than n. If m is missing altogether, then $m = n$ will be assumed.

That's it. An example:

```
*delete zeus3d.10,20
       a = b
       b = c
*insert mstart.100
       d(i,j,k) = 1.0
*i zeus3d.100
       c = d
*d zeus3d.120
```

Note that `*d` and `*i` are short forms for `*delete` and `*insert` respectively. In addition, `*replace` (`*rp` for short) is a synonym for `*delete`. In the example, lines 10 through 20 in the main program `zeus3d` are replaced with the two lines which set `a` and `b`, a single line

setting `d(i,j,k)` is inserted after line 100 in subroutine `mstart`, a single line setting `c` is inserted after line 100 in `zeus3d`, and line 120 in `zeus3d` is simply deleted.

To aid the user in deciding what changes to make and where to make them, a flow chart showing the sequence of the major subroutine calls in *ZEUS-3D* is given in App. A. This will be particularly useful once faced with the task of comprehending the source code listing, `dzeus35.n`.

If *EDITOR* detects any merge syntax errors or conflicts during the merge, it will write the merged file [as best as could be done given the error(s) detected] into a file named `dzeus35.m` and insert an error message immediately after each offending line. A merge error will prevent the second pass of preprocessing (*i.e.*, precompilation) from being executed and the user will be told what character pattern to search for in the file `dzeus35.m` in order to find the generated error messages.

Should the merge step be successful, *EDITOR* goes through a second pass and performs all the precompilation commands. These include:

1. `*if define,`*macro*—the following source code is kept provided the macro is defined by a `*define` statement somewhere in the file.

2. `*if -define,`*macro*—the following source code is kept provided the macro is *not* defined by a `*define` statement somewhere in the file.

3. `*if def,.not.`*macro*—same as 2. Note that `def` is an acceptable short form for `define`.

4. `*if def,`*macro1* `.and.` *macro2*—the following source code is kept provided both macros are defined by a `*def` statement somewhere in the file.

5. `*if def,`*macro1* `.or.` *macro2*—the following source code is kept provided either macro is defined by a `*def` statement somewhere in the file.

6. `*if alias` *macro*`.eq.`*phrase*—the following source code is kept provided the alias *macro* has been set to the character string *phrase* by an `*alias` statement somewhere in the file.

7. `*if alias` *macro*`.ne.`*phrase*—the following source code is kept provided the alias *macro* has *not* been set to the character string *phrase* by an `*alias` statement somewhere in the file.

8. `*else`—the following source code is kept if the truth value of the previous `*if` is false.

9. `*endif`—closes the previous `*if`, `*else` structure. All source code following the `*endif` statement is not affected by the previous `*if` or `*else` statements. For every `*if` statement, there must be an `*endif` statement which follows.

10. `*call` *deckname*—includes the contents of the common deck *deckname* at the location of the `*call` statement.

These precompiler commands can be used to construct the changes to be inserted into `dzeus35` using the *EDITOR* `*delete` and `*insert` commands. All changes should be placed in the user's change deck, which in our example, has been called `chguser`. These changes are then incorporated into the code by "de-commenting" the line `**read chguser` in the script file `dzeus35.s` by deleting one of the asterisks (§2.3).

Note that during both passes, the `*deck` and `*cdeck` statements are used as reference points, and are then expunged from the source code during the second pass. If any precompilation syntax errors are detected, *EDITOR* will write the precompiled file [as best as could be done given the error(s) detected] into a file named `dzeus35.f` and insert an error message immediately after each offending line. *EDITOR* will abort further processing and the user will be told what character pattern to search for in the file `dzeus35.f` in order to find the generated error messages. On the other hand, if the precompilation pass is successful, *EDITOR* will make yet another pass through the code to substitute `namelist` statements with subroutine calls, perform auto-tasking, update the files in the directory `dzeus3.5`, and create the makefile, `makezeus`. This makefile compiles only those subroutines affected by the changes made, links all the subroutines and libraries together, and creates the new executable `xdzeus35`.

A complete discussion of *EDITOR*'s `merge` and `precom`pilation features can be found in the *EDITOR* user manual `edit21_man.ps` found in the directory `manuals` of `dzeus35.tar` from `www.ica.smu.ca/zeus3d`.

5.3 Debugging in *ZEUS-3D*

It is the author's experience that virtually no change of significance can be introduced into the code without tripping up some problems requiring the debugger. And while the vast majority of these bugs will eventually be traced back to the user's initialisation routine or other change made by the user, it will often necessitate probing other parts of the code to find these problems. To someone not knowing their way around *ZEUS-3D*, this will come as a daunting task indeed. Therefore, this section attempts to offer—in a generic way—some guidance in starting a debugging session.

On virtually all *UNIX* platforms, the debugger `DBX` is available which allows the user to fire up `xdzeus35` (compiled with the `-g` option; see §2.3) within the debugging environment by typing:

```
dbx xdzeus35
```

at the *UNIX* prompt. From there the user can set "breakpoints", reassign variable values, and navigate pretty well anywhere within the code probing variable values as one moves along. For the uninitiated, a very short three-page primer on using `dbx`, `dprimer.ps`, may be found in the directory `manuals` of `dzeus35.tar` from `www.ica.smu.ca/zeus3d`.

While not specific to this package, the following discussion assumes `dbx` to be the default debugging environment.

1. Stop in your initialisation routine (`PROBLEM`)

The first task is to make sure all variables are set as you think they should be. Stop at the `return` statement of your initialisation routine, and probe all variable values, particularly

those around the periphery of the grid where users often forget to initialise the flow variables. 90% of all bugs a user introduces into the code can be traced to flow variables (density, velocity, *etc.*) not being assigned properly or fully. Make sure, for example, that each array is set from `1:in`, `1:jn`, and `1:kn` and not, for example, simply from `is:ie`, `js:je`, and `ks:ke` (see App. C for a review of the variable and parameter names and definitions). In addition, if any boundaries are set to "inflow" (`nflo=10`), the user will need to set the inflow boundary arrays such as `diib1`, `diib2`, `v1iib1`, *etc.* (see the template routine `myprob.f` in §5.1).

2. Stop in ***srcstep***

Stopping at the top of routine `srcstep` (where the source terms are updated; see the `dzeus35` "skeleton" in App. A for guidance on how to navigate through the code) will allow you to make certain that once set in the initialisation routine, all variables have been passed to the beginning of the first MHD cycle correctly. If execution dies before reaching `srcstep`, it is possible there has been a problem in one of the graphics routines, and you should probe the variable values there.

If execution makes it correctly to the top of `srcstep`, one can advance through `srcstep` one call after the other, making sure that each of `stv1`, `stv2`, `stv3`, `viscous`, *etc.*, is executing correctly.

3. Stop in ***trnsprt***

If `srcstep` reveals no anomolies, one next ventures into `trnsprt`, which takes care of the transport terms (fluxes), the induction equation, and the transverse Lorentz forces. Navigating through this routine is complicated by the fact that the order in which the constituent routines (`tranx1`, `tranx2`, `tranx3`, `cmoc1`, `cmoc2`, and `cmoc3`) are executed depends on how many MHD cycles have been run through. This is an attempt to reduce any favoritism among the directions in the directional- and planar-split algorithms.

The variable controlling the order in which the constituent routines are called is the integer `ix1x2x3` which can take on any integral value between and including 1 and 6. Knowing the value of `ix1x2x3` will tell you which segment of `trnsprt` (as well as `mom1`, `mom2`, and `mom3` should you have to venture there) you have to go to.

With any luck, you will not have to venture into the `cmoc*` routines, as these are long with many local variables, most of which are scalars. If the unfortunate has occured and you do need to know what values are being assigned to these local variables, it may be more convenient to use the "vectorised" versions of the `cmoc*` routines, namely `cmoc1v`, `cmoc2v`, and `cmoc3v` where all the local variables are at least defined as 1-D vectors, and thus can be probed at the end of the inner 1-D sweep and not within. Swapping the "scalar" versions for the vectorised versions of `cmoc*` is done most conveniently by going into the code `dzeus35` itself, and replacing:

```
*dk cmoc1
       subroutine cmoc1
*dk cmoc1v
       subroutine cmoc1v
```

with, respectively:

```
*dk cmoc1s
       subroutine cmoc1s
*dk cmoc1
       subroutine cmoc1
```

and *etc.* for `cmoc2` and `cmoc3`. This, of course, will require the code to be compiled again (*e.g.*, `csh -v dzeus35.s`). With these changes, the vectorised versions will now be called by `trnsprt` rather than the scalar versions. Alternately, you could just replace all the calls to `cmoc1`, *etc.*, in `trnsprt` with `cmoc1v`, *etc.*, where there are 18 such calls. Obviously, these changes are meant to be temporary and should be reset once your debugging session is complete.

4. Double-debug sessions

For the really stubborn bugs, often I have to do a "double debug session" in which I have `dzeus35` without the changes compiled and open in a debug session in one window on the left side of my screen, and the code with the changes opened up in a debug session in another window on the right side of my screen. From there, I undertake the tedious task of advancing through the code routine by routine, line by line until I find where the two versions diverge, and go from there.

5. "Gotchas"

Depending on your installation, `dbx` is capable of a number of very annoying "gotchas" which can slow progress markedly. I mention a few below.

While any variable declared either globally or local to the subroutine should, in principle, be accessible (*i.e.*, their values probed) from within the subroutine, this is not always the case. In some installations, only variables actually set or modified by the subroutine may be probed and, if you really need to see these variable values, one either has to go to a routine where these values are modifed or, if that is insufficient, put a "dummy" assignment statement in the routine (*e.g.*, `var = var + 0.0d0`), recompile, and restart the debug session.

In my own installation of `dbx`, local variables that are equivalenced in the subroutine to a globally declared variable are often inaccessible by the local variable name—one has to use the global variable name to probe values. This, like the first "gotcha", is a completely stupid design "feature", but you may be stuck with it. Sometimes using the global variable name is no big deal. However, in some situations such as in the `cmoc*` routines where the local variable is not dimensioned the same as the global array [*e.g.*, in `cmoc1`, local variable `v2t(kn,jn)` is equivalenced to `wk2d(jn,kn)` in 3-D], it isn't always so simple to determine the indices needed for the global array to retrieve the desired element of the local array. In the example given, it is not a simple matter of just swapping the indices `j` and `k` when `jn`$\neq$`kn`. In these situations, it may be necessary to comment out the equivalence statements and recompile, hoping that this doesn't somehow affect the nature of the problems you are trying to uncover.

Finally, `dbx` will only report the first 15 decimal places of the variable (for double precision), and sometimes problems start to occur in the 16th decimal place or beyond. Even though values beyond the 15th or 16th significant figure are normally considered "noise", unlike real noise computer noise is always repeatable and thus can serve as a useful indica-

tor of when deviation from the correct answer begins. In `dbx`, for example, one can probe these additional digits by subtracting the reported result from the variable itself. Thus, if `v1(i,j,k)` is reported as `3.14159265358979`, you could type :

```
print v1(i,j,k)-3.14159265358979
```

which may, for example, then report:

```
v1(i,j,k)-3.14159265358979 = 2.5430056384790e-16
```

which can be compared to another session to make sure differences aren't creeping in at this extremely low but sometimes significant level. It is my experience that if `dbx` reports two number to be equal to 15 decimal places, they aren't always equal. However, if the noises are also equal then the variable values can safely be taken as identical.

These are only a few general guidelines to debugging within `dzeus35`, and may cover 95% of the situations a typical user may encounter. If bugs or "undesired features" are found or strongly suspected in the code itself separate from changes introduced by the user, the user is encouraged to report these to the *ZEUS* forum accessible from `http://ica.smu.ca/zeus3d` with as much description as possible, including the `.mac` file, the `.s` file, and any change deck as may be appropriate. One of the developers or users of the code may have a work-around and, if significant, an attempt will be made to address the problem by the next release.

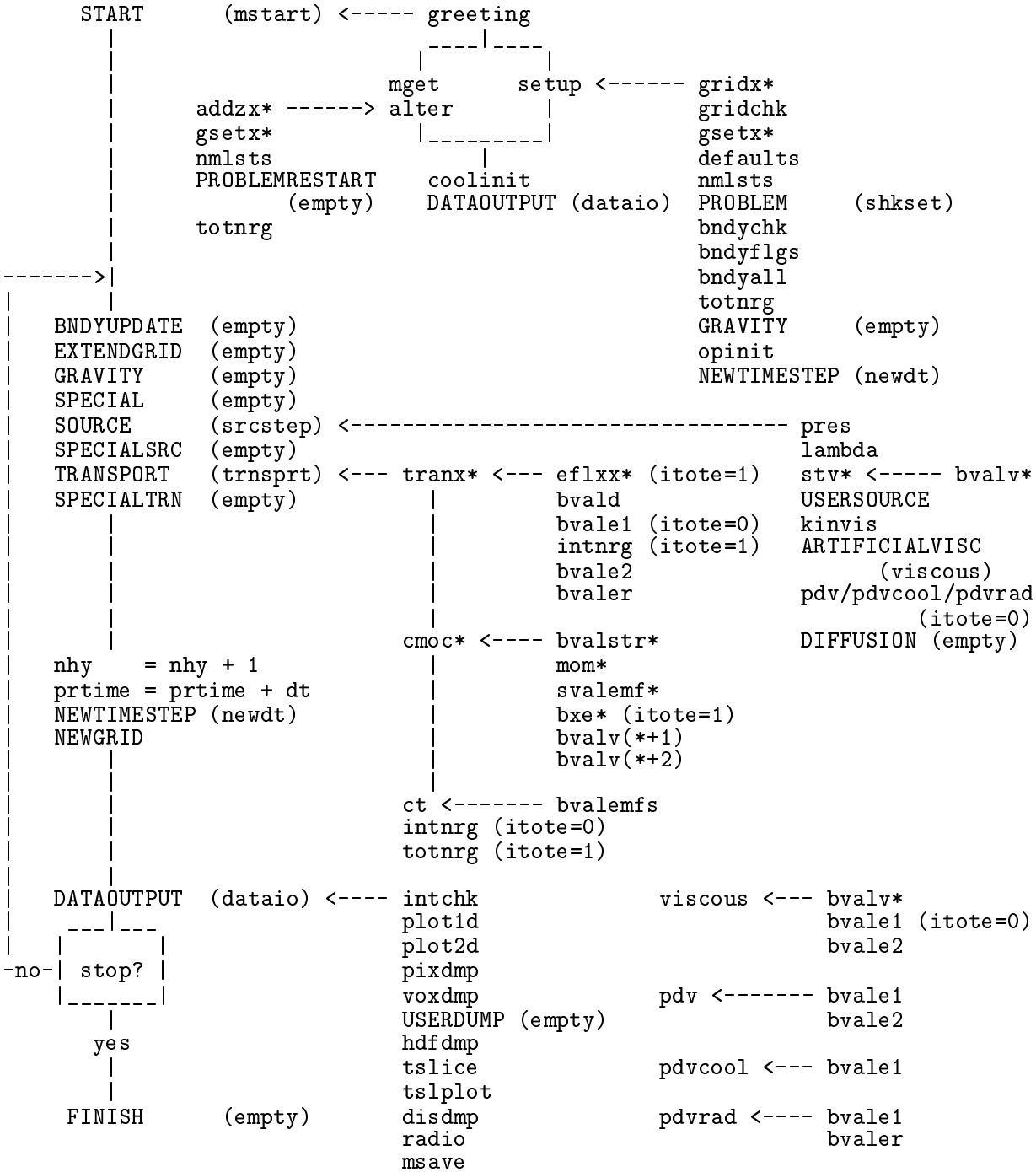
6 Quick summary

This final section is intended to serve as a quick reference sheet for those who are already familiar with running *ZEUS-3D*.

1. Set the macros in the file `zeus35.mac` (§2.2 and App. A).
2. Make the necessary changes to the `dzeus35.s` script file, including the parameters in the change deck `chgzeus` (§2.3.3) and the input parameters in the input deck `inzeus` (§2.3.5 and App. B).
3. Put the desired source code changes, if any, into the file `chguser` (§5), and "de-comment" the line `**read chguser` in the script file `dzeus35.s` by deleting one of the asterisks (§2.3).
4. Run the script file to create the *ZEUS-3D* executable by typing `csh -v dzeus35.s`
5. Fire up the executable by either typing `xdzeus35`, or by submitting the job to the appropriate batch queue.

A The *ZEUS-3D* skeleton

Modules in upper case are *EDITOR* aliases, set in `zeus35.mac`. Modules in lower case are actual subroutines in the source code. An asterisk (*) in a subroutine name is a "wild-card" for 1, 2, and 3. Exemplary choices for the *EDITOR* aliases are given parenthetically, and are appropriate for the 1-D Brio and Wu MHD shock tube problem for which the sample files `zeus35.mac` (§2.2) and `dzeus35.s` (§2.3) were designed. Additional choices for the *EDITOR* module name aliases appearing in the skeleton below are listed on the next page.



START	mstart	standard initialisation of variables
BNDYUPDATE	empty	
	breset	to reset flow-in boundary values, used in test problems
	wiggle	to wiggle jet inlet
	bgen	to generate magnetic field at jet inlet
	jetbndy	calls both subroutines `wiggle` and `bgen`
EXTENDGRID	empty	
	extend	to extend computational domain
GRAVITY	empty	no self-gravity
	gravity	one of two Poisson solver algorithms may be chosen
SPECIAL	empty	
	...	user-defined module for additional physics
SOURCE	empty	for advection tests
	srcstep	standard source term module
SPECIALSRC	empty	
	...	user-defined module for additional source terms
TRANSPORT	empty	
	trnsprt	standard transport module
SPECIALTRN	empty	
	resetv	for advection tests
	...	user-defined module for additional transport terms
NEWTIMESTEP	newdt	full dynamics
	advectdt	for advection tests
DATAOUTPUT	empty	
	dataio	standard I/O module
FINISH	empty	
	...	user-defined module called once at the end of execution
USERSOURCE	empty	
	phistv	non-conservative point mass gravity (see `corona`)
	...	user-defined module for additional source terms
ARTIFICIALVISC	empty	
	viscous	von Neumann-Richtmyer artificial viscosity
	gasdiff	heat and mass diffusion
DIFFUSION	empty	
	diffuse	second fluid diffusion
USERDUMP	empty	
	...	user-defined I/O module
PROBLEM	shkset	for shock tube tests
	corona	sets up jet-disc problem
	jetinit	sets up propagating jet problem
	...	numerous others already in the code
	...	user-defined module to initialise flow variables
PROBLEMRESTART	empty	
	...	user-defined module to alter variables for restarted job

B The namelists

There are some 500 `namelist` parameters to specify a unique initialisation. Take heart—most defaults can be used for most applications. As a start, use the input deck given in the `dzeus35.s` template (§2.3), and then alter as needed.

On the next page begins a complete catalogue of all the input parameters in `dzeus35`. The parameters are grouped together in "namelists" and discussion for each namelist is contained within a segment headed by the name of the namelist and the subroutine in which the namelist is called. For example, the first namelist is `iocon` (input/output control) and is called by the subroutine `mstart`. After each heading is a discussion of what the namelist controls, a list of all the parameters which are elements of the namelist, and finally the syntax used in `dzeus35` to declare the namelist.

For the uninitiated, `namelist` is a non-standard feature of most *FORTRAN77* compilers and a standard feature of *FORTRAN90* which provides a convenient way to specify input data. Before *FORTRAN90* was released in 1994, each platform had its own `namelist` with its own syntax, and this made it difficult to port *ZEUS-3D* even among different flavours of *UNIX*. Thus, a `namelist` emulator was built into *EDITOR* which, during one of its many passes through the code, replaces all namelist references (including `read`s and `write`s) with calls to subroutines in the `dnamelist.a` library. The following discussion, therefore, reflects the syntax for the *EDITOR* `namelist`, which differs somewhat from the *FORTRAN90* version. If desired, *EDITOR* can be instructed not to replace the `namelist` syntax (`inmlst=0`), in which case your compiler's `namelist` would be invoked. This may cause syntax errors to be issued since standard *FORTRAN* `namelist`s don't allow variables passed via a subroutine to be used as a namelist parameter, whereas the *EDITOR* `namelist` does.

In order to specify an input parameter, one merely needs to set it to the desired value as done in the input deck `inzeus` found in the sample script file `dzeus35.s` (§2.3). The order in which the variables appear in the namelist declaration need not be adhered to in the input deck nor must all the variables be set. So long as the variable specified in the input deck is a member of the namelist, then `namelist` will set the variable as specified.

There are a few rules to bear in mind. The namelists (but not necessarily the namelist variables) in the input deck must be in the same order as they are encountered during execution. If no parameters are to be set, an empty namelist (one with the namelist name between two $ sentinels) must appear in the correct sequence. There is no problem with namelists appearing that are never read, but a read to a non-existent namelist will generate a `namelist` error message. In this catalogue, the order of the namelists is the same as the order in which they appear in `inzeus` and in which they are encountered in `dzeus35`.

The syntactic rules of setting the variables can be gleaned from the input deck `inzeus` (§2.3). Column 1 is reserved for a '`c`' to "comment out" a namelist line which is then echoed on the CRT when encountered in the input deck. Column 2 is reserved for the leading $ sentinel. The specification of the namelist may start in column 3 and must terminate with a second $ sentinel. Until the second $ sentinel is found, all lines will be interpreted as part of the same namelist. All characters appearing after the 72nd column will be ignored, including the closing $ sentinel, should it inadvertently be placed there.

B.1 IOCON—I/O CONtrol (subroutine MSTART)

This namelist sets the logical units to be used during execution. Typically, these parameters will not need to be set to anything other than their default values. These parameters are *not* written to the restart dump. Therefore, all non-default values for any of the parameters in this namelist must be set each time the job is resumed.

```
parameter                    description                          default

iotty      logical unit for terminal (standard output)              6
ioplt1     logical unit for 1-D plots using NCAR/PSPLOT graphics   99
ioplt2     logical unit for 2-D plots using NCAR/PSPLOT graphics   99
iolog      logical unit for message log dump                       30
iodmp      logical unit for restart dumps                          31
iopix      logical unit for pixel dumps                            32
iousr      logical unit for user dumps                             33
iotsl      logical unit for time slice (history) ascii dumps       34
iotslp     logical unit for time slice (history) plot  dumps       99
iovox      logical unit for voxel dumps                            35
iodis      logical unit for display dumps                          36
iorad      logical unit for RADIO dumps                            37

WARNING: AVOID LOGICAL UNIT 3. APPARENT CONFLICT WITH NCAR.

NOTE   : IOTTY MAY BE SET TO 6 (TO GET CRT OUTPUT) OR 0 (NO OUTPUT).

      namelist / iocon    /
     1               iotty   , ioplt1  , ioplt2  , iolog   , iodmp
     2             , iopix   , iousr   , iotsl   , iotslp  , iovox
     3             , iodis   , iorad
```

B.2 RESCON—RESstart dump CONtrol (subroutine MSTART)

This namelist determines if the job is to be started from initial conditions, or if it is to be restarted from a previous run. These parameters are *not* written to the restart dump. Therefore, all non-default values for any of the parameters in this namelist must be set each time the job is resumed.

The default values are set for starting from initial conditions, which occurs when the third to fifth characters in `resfile` are `000`. To restart a job, one can usually use the same input deck as was used for the original run with `resfile` set to the desired restart dump name. In addition, parameters in the namelist `pcon` may have to be changed.

The parameters `*getm?`; `*=i,j,k`, `?=n,x` are designed so that only a portion of the restart dump may be read, and/or so that the data may be read into a larger grid. That is, it is not necessary for the field arrays in a restarted job to be dimensioned the same as those in the run which generated the restart dump.

Example 1: For a straight restart without altering the grid or the *EDITOR* macros, leave the values of `igetmn`, *etc.* to their defaults and make sure that the parameters `in`, *etc.* are set to the same values as in the run which generated the restart dump.

Example 2: If the first run was on a 64^3 grid and the user wishes to read the inner eighth of

the data and position them at the centre of a 100^3 grid, and if the new portion of the grid is to be determined from the existing grid, then the following settings are necessary:

```
igetmn = 17, jgetmn = 17, kgetmn = 17, iaddz = 1
igetmx = 48, jgetmx = 48, kgetmx = 48, jaddz = 1
iputmn = 35, jputmn = 35, kputmn = 35, kaddz = 1
```

The desired portion of the restart dump will be read and loaded into the 100^3 grid between `i=35,66`, `j=35,66`, `k=35,66`. In addition, the 1-grid `x1a(35:66)` (see §C.1 for a discussion of the naming convention for the grid variables) will be filled by the values of `x1a(17:48)` in the restart dump. The code will detect that the grids `x1a`, `x2a`, `x3a` are now incomplete, and will call the appropriate modules to add zones to the `x1`-, `x2`-, and `x3`-grids. If the user wishes, (`*addz=1`, `*=i,j,k`), the new portion of the grid may be determined automatically from the existing grid. In this example, `x1a(1:34)` would be determined (*i.e.*, `dx1min`, `x1rat`, *etc.*, see namelist `ggen1`) from `x1a(35:37)`. Similarly, `x1a(67:100)` would be determined from `x1a(64:66)`. Alternatively, the user may opt to set the new portion of the grid manually. In this case, one should set `*addz=0` and proceed with setting the namelists `ggen1`, `ggen2`, `ggen3`. (See discussion in `ggen1`.) Note that if the user selects the manual option, it is imperative that the portion of the new grid that overlaps the old grid be, in fact, identical to the old grid. Next, all arrays will be padded with values at the edges of the portion read. Thus `d(1:34,j,k)=d(35,j,k)`, `d(67:100,j,k)=d(66,j,k)` (where `d` is the density array—see §C.2), *etc.* Of course, the user is free to set the values of the padded portion of the arrays to whatever values they want by linking a user-supplied subroutine to the *EDITOR* macro `PROBLEMRESTART` (§2.2.2).

Finally, a job may be resumed from a restart dump with different *EDITOR* macros defined or not. Thus, if a job that began with magnetic fields is to be resumed without them, the user may recompile `dzeus35` *without* magnetic fields (`MHD` not defined) and then blindly read the restart dump which contains magnetic field arrays. There is enough information in the restart dump that the code can selectively read the non-magnetic part of the dump and resume the calculation as though there were never any magnetic fields. Of course, whether suddenly disappearing the magnetic fields is physically realistic is another matter!

parameter	description	default
dtdmp	problem time interval between restart dumps = 0 => no restart dumps (probably a bad idea) > 0 => write each dump to a new file < 0 => overwrite old even (odd) numbered dump with new even (odd) numbered dump at time interval abs(dtdump)	0.0
nresdmp	the sequential number for the next restart dump < 0 => nresdmp = iresdmp	-1
nlogdmp	the sequential number for the next log file < 0 => nlogdmp = ilogdmp	-1
idtag	character*2 problem tag appended to filenames	'aa'
resfile	restart dump filename	'zr000aa'
igetmn	minimum x1-index (i) to be read from restart dump	1
igetmx	maximum x1-index (i) to be read from restart dump	in
iputmn	i-index at which x1a(igetmn) is stored	1

```
iaddz      < 0 => no new zones are generated                          0
           = 0 => call GRIDX1 to redo entire grid
           > 0 => new zone spacing determined from existing grid
```

The variables (jgetmn, jgetmx, jputmn, jaddz) and (kgetmn, kgetmx, kputmn, kaddz) are analogous to (igetmn, igetmx, iputmn, iaddz) for the 2- and 3-directions respectively.

```
      namelist / rescon   /
     1               dtdmp   , nresdmp , nlogdmp , idtag   , resfile
     2             , igetmn  , igetmx  , jgetmn  , jgetmx  , kgetmn
     3             , kgetmx  , iputmn  , jputmn  , kputmn  , iaddz
     4             , jaddz   , kaddz
```

B.3 GGEN1—Grid GENerator for x1 (subroutine GRIDX1)

This namelist controls how the grid is determined in the 1-direction. All the parameters in this namelist, as well as those in namelists ggen2, ggen3, and those read by subroutine nmlsts are written to the restart dump. The stored values, therefore, will become the "default" values of the parameters for any run resumed from the restart dump.

The grid can be created all at once or in several blocks. Each block requires a separate read of this namelist specifying how that portion of the grid is to be computed. The parameter lgrid should be set to .true. (or equivalently .t. for the *EDITOR* namelist) only for the last block. (Note that the *EDITOR* namelist also allows .f. as a short form for .false..)

There are two types of gridding. The first is "ratioed gridding" where the distance across a zone is a fixed multiple of the distance across the previous zone. If this multiple is 1, then the zones are uniform. If the multiple is 1.1, then each zone is 10% larger than the previous one. If the multiple is 0.9, then each zone is 10% smaller than the previous one. To determine a block of ratioed zones uniquely, one must specify the number of zones in the block (nbl), the minimum and maximum extent of the block in coordinate units (x1min, x1max), and EITHER the smallest zone size in the block (dx1min) OR the ratio to use between zones (x1rat). Specifying either dx1min or x1rat will allow the other to be computed.

The second type of gridding is "scaled gridding" where the coordinate value is some fixed multiple of the previous coordinate value. For ratioed grids, dx(n)=mult*dx(n-1). For scaled grids, x(n)=mult*x(n-1). For example, scaled gridding would be appropriate for the r-direction in spherical polar coordinates if the zones were all to have the same *shape*. To determine a block of scaled zones uniquely, one must specify the number of zones in the block (nbl) and the minimum and maximum extent of the block in coordinate units (x1min, x1max). Neither dx1min nor x1rat are needed.

The grid can be scaled to physical units most conveniently by setting the multiplicative factor x1scale to the desired scaling value.

For restarted jobs, there is a third gridding option. Setting igrid to zero will cause the grid generator to skip over the nbl zones specified for this block. Thus, in the second example in the discussion for namelist rescon, one could set the new zones for the x1-direction manually with three ggen1 namelist "cards". The first card would set zones (1:34) in whatever manner desired with the condition that the last zone of the new grid ends where the first zone of the old grid begins. The second card would set igrid=0 and nbl=32. This

would leave zones (35:66) alone since they were set when the restart dump was read. Finally, the third card would set zones (67:100) in whatever manner desired with the condition that the first zone of the new grid begins where the last zone of the old grid ends.

Other than remaining within the memory limits of the machine, there are two practical considerations when choosing the number of zones for each of the three dimensions. First, if at all possible, the greatest number of zones should be along the 1-direction so that the vector length of the vectorised loop is as long as possible[4]. Second, if the code is to be multi-tasked, the number of zones (including the five boundary zones) in each direction should be an integral multiple of the number of parallel processors available on the machine. This will yield the best overall degree of parallelism.

```
parameter                    description                          default

nbl        number of active zones in block being generated          1
x1min      x1a(imin); bottom position of block                      0.0
x1max      x1a(imax); top    position of block                      0.0
x1scale    arbitrary scaling factor for "x1min" and "x1max"         1.0
igrid      method of computing zones.                               1
           = 0  => block has already been set (restarted runs only)
           =+1  => (ratioed) use input "x1rat" to compute "dx1min".
                   "dx1min" = size of first zone in block
           =-1  => (ratioed) use input "x1rat" to compute "dx1min".
                   "dx1min" = size of last  zone in block
           =+2  => (ratioed) use input "dx1min" to compute "x1rat".
                   "dx1min" = size of first zone in block
           =-2  => (ratioed) use input "dx1min" to compute "x1rat".
                   "dx1min" = size of last  zone in block
           = 3  => (scaled) compute "x1rat" and "dx1min" from "nbl".
x1rat      desired ratio dx1a(i+1) / dx1a(i)                        1.0
dx1min     size of first (igrid>0) or last (igrid<0) zone in block  0.0
lgrid      =.false. => read another block (namelist card).        .false.
           =.true.  => all blocks are read in.  Do not look for
                       another "ggen1" namelist card.

      namelist / ggen1    /
    1               nbl     , x1min   , x1max   , x1scale , igrid
    2             , x1rat   , dx1min  , lgrid
```

B.4 GGEN2—Grid GENerator for x2 (subroutine GRIDX2)

See comments for GGEN1.

```
parameter                    description                          default

nbl        number of active zones in block being generated          1
x2min      x2a(jmin); bottom position of block                      0.0
x2max      x2a(jmax); top    position of block                      0.0
x2scale    arbitrary scaling factor for "x2min" and "x2max"         1.0
igrid      method of computing zones.                               1
           = 0  => block has already been set (restarted runs only)
           =+1  => (ratioed) use input "x2rat" to compute "dx2min",
                   "dx2min" = size of first zone in block
```

[4]This is an issue only if one is using a vector machine.

```
           =-1  => (ratioed) use input "x2rat" to compute "dx2min",
                   "dx2min" = size of last  zone in block
           =+2  => (ratioed) use input "dx2min" to compute "x2rat",
                   "dx2min" = size of first zone in block
           =-2  => (ratioed) use input "dx2min" to compute "x2rat",
                   "dx2min" = size of last  zone in block
           = 3  => (scaled) compute "x2rat" and "dx2min" from "nbl".
x2rat      desired ratio dx2a(j+1) / dx2a(j)                          1.0
dx2min     size of first (igrid>0) or last (igrid<0) zone in block    0.0
units      sets the angular units (character*2, RTP only)            'rd'
           'rd' => radians, 'pi' => pi radians, 'dg' => degrees
lgrid      =.false. => read another block (namelist card).         .false.
           =.true.  => all blocks are read in.  Do not look for
                       another "ggen2" namelist card.

      namelist / ggen2    /
    1               nbl     , x2min   , x2max   , x2scale , igrid
    2             , x2rat   , dx2min  , units   , lgrid
```

B.5 GGEN3—Grid GENerator for x3 (subroutine GRIDX3)

See comments for GGEN1.

```
parameter                      description                          default

nbl        number of active zones in block being generated            1
x3min      x3a(kmin); bottom position of block                        0.0
x3max      x3a(kmax); top    position of block                        0.0
x3scale    arbitrary scaling factor for "x3min" and "x3max"           1.0
igrid      method of computing zones.                                 1
           = 0  => block has already been set (restarted runs only)
           =+1  => (ratioed) use input "x3rat" to compute "dx3min",
                   "dx3min" = size of first zone in block
           =-1  => (ratioed) use input "x3rat" to compute "dx3min",
                   "dx3min" = size of last  zone in block
           =+2  => (ratioed) use input "dx3min" to compute "x3rat",
                   "dx3min" = size of first zone in block
           =-2  => (ratioed) use input "dx3min" to compute "x3rat",
                   "dx3min" = size of last  zone in block
           = 3  => (scaled) compute "x3rat" and "dx3min" from "nbl".
x3rat      desired ratio dx3a(k+1) / dx3a(k)                          1.0
dx3min     size of first (igrid>0) or last (igrid<0) zone in block    0.0
units      sets the angular units (character*2, ZRP and RTP only)    'rd'
           'rd' => radians, 'pi' => pi radians, 'dg' => degrees
lgrid      =.false. => read another block (namelist card).         .false.
           =.true.  => all blocks are read in.  Do not look for
                       another "ggen3" namelist card.

      namelist / ggen3    /
    1               nbl     , x3min   , x3max   , x3scale , igrid
    2             , x3rat   , dx3min  , units   , lgrid
```

B.6 PCON—Problem CONtrol (subroutine NMLSTS)

Determines the criteria for terminating the job.

```
parameter                       description                                  default

nlim       cycles to run                                                       0
tlim       physical (problem) time to stop calculation                         0.0
           if tlim < 0, problem is stopped at exactly abs(tlim)
ttotal     total seconds of execution time permitted for job                   0.0
tsave      seconds of execution (cpu) time reserved for cleanup                0.0

      namelist / pcon     /
     1               nlim    , tlim    , ttotal  , tsave
```

B.7 HYCON—HYdro CONtrol (NMLSTS)

Sets the parameters which control the hydrodynamics. One of the most important selectors in this namelist, `itote`, chooses between the internal (`itote=0`) and total (`itote=1`; default) energy equations. This is the first release of *ZEUS-3D* with a complete installation of the total energy equation and is *generally* found to give superior results. *Pros* and *cons* for choosing between the two energy equations include: Execution time is generally faster for the internal energy equation (by about 20%), and pressures are guaranteed positive definite for `courno<0.5`. However, the algorithm is not strictly conservative which, among other things, causes it to converge on incorrect values (by as much as 20%) in some 1-D shock-tube tests. The total energy equation is conservative and somewhat more stable allowing a slightly larger Courant number (*e.g.*, `courno=0.75`) than the internal energy equation in some applications. However, internal energies are not positive definite and where they become negative, are reset to `e1floor`. In the opinion of this author, the requirement that a code be strictly conservative has been somewhat overblown in the literature. It is true that only conservative codes will converge correctly on 1-D shock tube problems but, in the messy universe where the sum of mechanical and thermal energies is known not to be conserved, a strictly conservative code may less important than the assurance of positive-definite pressures. This version of the code gives the user both options.

All energy variables should be interpreted as *energy per unit volume*. In setting up a problem, the user should always initialise the internal energy density (variable `e1` and boundary values `e1iib1`, *etc.*; see §C.2 and §C.3) and not the total energy density, (`et`), regardless of (`itote`). Being a primitive variable, boundary conditions are always applied to the internal energy density. Note that if `ISO` is defined, `itote` is set to 0.

The steepest discontinuities this code can sustain are obtained with `iord=2`, `iords=3`, and `istp=2`, where the latter assures only *contact* discontinuities are steepened. `istp=1` will cause *any* discontinuity to be steepened and is intended for advection tests only. These settings maintain contacts in 2 or 3 zones and most shocks in `qcon+1` zones (although some slow shocks may be smeared out over as many as ten zones), but can also can cause the bases of discontinuities or rarefactions to undershoot slightly and even "ring" in some 1-D shock tube tests. More conservative settings are the defaults, for which the code runs 20% faster and maintains contacts in 5 to 7 zones, and most shocks to `qcon+2` zones.

```
parameter                       description                                  default

qcon       quadratic artificial viscosity (q) constant                         2.0
qlin       linear    artificial viscosity (q) constant                         0.0
```

```
courno     Courant number                                              0.5
dtrat      ratio of "dtmin" to initial value of "dt"                   0.001
dtmax      maximum time step to use                                    huge
iord       order of interpolation                                       2
           Legal values are 1 (donor cell), 2 (van Leer),
           -2 (velocity-corrected van Leer), 3 (ppi)
iords      order of interpolation for scalars to override "iord"       iord
istp       contact discontinuity steepener (third order only)           0
           0 => always off, 1 => always on,  2 => on only at
           contact discontinuity
**floor    smallest value desired for variable **          scalars  tiny
                                                           vectors  0.0
icool      0 => use PDV in SRCSTEP                                      0
           1 => use PDVCOOL in SRCSTEP for pdv work with arbitrary
                cooling function
itote      0 => solve the internal energy equation (positive            1
                definite pressures but energy not conserved).
           1 => solve the total energy equation (energy conserved
                but pressure not positive definite)
iscydf     0 => no subcycling on diffusion                              0
           1 => subcycle on diffusion
iscyqq     0 => no subcycling on artificial viscosity                   0
           1 => subcycle on artificial viscosity (itote=0 only)
ix1x2x3    seed for directional splitting sequence                      1
mind       minimum value subroutine MINDEN will allow for density    dfloor
nu         kinematic viscosity (in units of LV)                       0.0
isetemf    affects flow-in skin values for emf(perp) and flow-in        0
           boundary values for emf(par)
           =0 => SVALEMF* and BVALEMFS don't overwrite (C)MOC*-
                 computed flow-in emfs with pre-set flow-in arrays.
           =1 => SVALEMF* overwrite (C)MOC*-computed flow-in emfs with
                 preset flow-in arrays, but BVALEMFS doesn't.
           =2 => Both SVALEMF* and BVALEMFS overwrite (C)MOC*-computed
                 flow-in emfs with preset flow-in arrays.
tspinup    time to add all the desired angular velocity (SPINUP).      1.0
           1.0 is characteristic time scale of the Bondi problem.
delta      amplitude of imparted angular velocity (SPINUP)        sqrt(2.0)
           Default puts centrifugal barrier at critical point
           (r=1) for perturbed Bondi flow.
orbchk     =0 => BNDYCHK will abort at a non-physical boundary.         0
           =1 => BNDYCHK issues a warning at a non-physical
                 boundary, but execution continues (dangerous!).
```

The routine SPINUP and the associated namelist variables tspinup and delta were designed to perturb Bondi flow to form discs, but can be used in other applications in which a gradual spin-up of the grid is desired.

```
      namelist / hycon    /
     1              qcon    , qlin    , courno  , dtrat   , dtmax
     2            , iord    , iords   , istp    , dfloor  , efloor
     3            , e2floor , v1floor , v2floor , v3floor , b1floor
     4            , b2floor , b3floor , icool   , itote   , iscydf
     5            , iscyqq  , ix1x2x3 , mind    , nu      , isetemf
     6            , tspinup , delta   , orbchk
```

B.8 IIB—Inner I Boundary control (NMLSTS)

This namelist specifies both the boundary type and the inflow values for the variables that can be set at the inner-i boundary. These variables are *not* declared if the *EDITOR* macro `ISYM` is set. Any one of ten MHD boundary conditions may be specified independently at every boundary zone by setting `niib(j,k)` to the desired value of `btype`, as follows:

```
btype =  1 (-1) =>  reflecting; grid singularity or symmetry axis
      =  2  (1) =>  reflecting; non-conducting boundary
      =  3  (5) =>  reflecting; conducting boundary
      =  4  (6) =>  reflecting; B continuous across boundary
      =  5  (4) =>  periodic
      =  6      =>  self-computing (for AMR)
      =  7      =>  outflow (not yet functional)
      =  8      =>  selective inflow
      =  9  (2) =>  non-characteristic outflow
      = 10  (3) =>  non-characteristic inflow
```

where the values of `btype` used in all previous versions of *ZEUS-3D* are given parenthetically.

The boundary values for the variables are used only in the event that a zone along the boundary is inflow (`btype=8,10`). Otherwise, the boundary value is determined from the flow variables on the active portion of the computational grid. The flow variables are `d` (density), `e1` (first internal energy density), `e2` (second internal energy density), `er` (radiation energy density), `v1` (1-velocity), `v2` (2-velocity), and `v3` (3-velocity). In addition, skin values for the transverse *emf* components and boundary values for the transverse magnetic field components can be set (see extensive discussion below).

The boundary type for the gravitational potential (`gtype`) is treated independently of the MHD boundaries, since the nature of the Poisson equation (elliptical) is different from that of the MHD equations (hyperbolic). Gravitational boundary type is specified by setting `giib` to the desired value of `gtype`, as follows:

```
gtype = 5  (4) =>  periodic
      = 9  (2) =>  six-term multipole expansion
      = 10 (3) =>  analytical (or preset) boundary values stored in gpiib.  Time-
                   varying boundaries can be updated by a routine aliased to
                   BNDYUPDATE
```

where the values of `gtype` used in all previous versions of *ZEUS-3D* are given parenthetically. Any other value for `gtype` means the boundaries of ϕ are never updated, which would be appropriate for constant boundary values set as part of the initial conditions.

NEW TO VERSION 3.5: Magnetic boundary conditions have been completely revamped, and a new, stable algorithm has been implemented. To start, a distinction is now made between the *skin* (which can receive characteristic information directly from the boundary region and/or the active grid) and the *boundary*, which can only receive characteristic information from the boundary region within a given CFL-limited timestep. For example, at the inner-i boundary, the `i=is` face constitutes the "skin", while all zones—face- or zone-centred—at `i=ism1, ism2` are *in* the boundary. Because of this distinction, magnetic "skin" values and "boundary" values are now treated differently in *ZEUS-3D*. Skin values are set in

routines `SVALEMF*` which are called by the `(C)MOC*` routines where the distinction between the two "terms" in the *emf*s is needed to set some skin conditions. Boundary values are set by `BVALEMFS` called at the top of `CT` where all components of the *emf*s are needed to set the boundary conditions for each.

The main problem with the algorithm used in version 3.4 was that skin values of the magnetic field were set directly when setting boundary conditions. This is folly, since resetting the magnetic flux through the face of a zone lying along the skin changes the net magnetic flux into the adjacent grid zone, introducing a magnetic monopole. This problem was particularly acute when setting inflow boundaries, and thus all inflow arrays such as `b1iib1` which allowed the normal (to the boundary) magnetic field to be set directly both on the skin and inside the boundary have been purged from the code.

In this version, the user must now completely initialise all magnetic field arrays (`b1`, `b2`, and `b3`) in their problem generator, including all boundaries. For inflow boundaries, *skin* values of the parallel field (*e.g.*, B_1 at the i-skins) are maintained by user-set arrays `emf2iib1` and `emf3iib1` which lie along the inner i-skin. Typically, these components of the *emf* are set by physical boundary conditions on the skin (*e.g.*, $v_2 = v_3 = 0$; $B_2 = B_3 = 0 \Rightarrow \varepsilon_2 \propto v_3 B_1 - v_1 B_3 = 0$; $\varepsilon_3 \propto v_1 B_2 - v_2 B_1 = 0$). Meanwhile, *boundary* values for the parallel field [*e.g.*, `b1(ism2:ism1)`] are determined by the solenoidal condition and thus are allowed to "float", regardless of boundary conditions. Accordingly, there are no arrays `emf2iib2`, `emf3iib2`, *etc.*

As in previous versions of the code, inflow conditions on the *transverse* magnetic field components (*e.g.*, B_2 and B_3 at the i-boundaries) are controlled by the user-set arrays `b2iib1`, `b2iib2`, `b3iib1`, and `b3iib2` at the inner-i boundary. Note that there are no *skin* values for the transverse components, only boundary values. Should constant boundary values be desired, these are most conveniently set to the corresponding initial values of `b2` and `b3`. Should these components need to vary in time, the user must supply updated values of `b2iib1`, `b2iib2`, `b3iib1`, and `b3iib2` at the current time and proper location at the beginning of each MHD cycle. For this, the *EDITOR* alias `BNDYUPDATE` can be aliased to a user-supplied routine that sets the boundary arrays as needed (§2.2.2).

Further, the concept of *selectively* setting inflow conditions depending on whether various characteristics arrive at the skin from the boundary or grid has been introduced. For the *emf*s, selective inflow conditions is controlled by the parameter `isetemf`, set in namelist `hycon`. In general, if one expects the skin to receive information only from the boundary region (as in *super-fast* inflow), `isetemf` should be set to 1 and the user should supply values for the transverse *emf*s based on boundary conditions alone. If flow across the boundary is *sub-fast*, *sub-Alfvén*, or even *sub-slow*, `isetemf` *probably* should be set to 0 (in which case, the skin *emf*s will remain as computed by the `(C)MOC*` routines), though there are circumstances where it still may be best to set `isetemf` differently (*e.g.*, `CORONA`). Regardless, care must be taken not to overspecify the boundary.

Selective inflow conditions are enabled using the new boundary type 8. This behaves just like boundary type 10 (inflow), except it allows designated variables to "float" at the boundary (take on the nearest grid value) rather than being set to the pre-determined boundary arrays (*e.g.*, `diib1` for density). To specify that a variable float, its boundary array should be set to the global parameter `huge` (§C.6), a nonsensical value that triggers logic in the boundary routines to set the variable according to its value in the nearest active grid zone

(very similar to reflecting boundary type 2, except `v1` is not set to zero at `i=is`). An example of the use of selective inflow conditions may be found in the problem generator `CORONA` for the case where local parameter `oork=2`. This establishes the so-called *Krasnopolsky conditions*, first introduced into an NCSA version of *ZEUS-3D* by Krasnopolsky, *et al.* (1999, ApJ, 526, 631) to launch sub-Alfvénic (initially) astrophysical jets from an accretion disc maintained as a boundary condition.

Additional discussion may be found in §1.5.

```
parameter                      description                          default

niib  (j,k)     "btype" of inner i boundary on sweep j,k             9
giib            "gtype" of entire inner i boundary                   2
**iib1(j,k)     first  inner i boundary value of variable **      **floor
                for sweep j,k (flow in only)
**iib2(j,k)     second inner i boundary value of variable **      **floor
                for sweep j,k (flow in only)
**iib3(j,k)     third  inner i boundary value of variable **      **floor
                for sweep j,k (flow in only)
gpiib (j,k)     analytical or preset values for gp on iib          0.0
                for sweep j,k (giib=3 only)
```

```
       namelist / iib      /
     1               niib    , giib    , diib1   , diib2   , v1iib1
     2             , v1iib2  , v1iib3  , v2iib1  , v2iib2  , v3iib1
     3             , v3iib2
*if -def,ISO
     4             , e1iib1  , e1iib2
*endif -ISO
*if def,TWOFLUID
     5             , e2iib1  , e2iib2
*endif TWOFLUID
*if def,RADIATION
     6             , eriib1  , eriib2
*endif RADIATION
*if def,GRAV
     7             , gpiib
*endif GRAV
*if def,MHD
     8             , emf1iib1, emf1iib2, emf2iib1, emf3iib1, b3iib1
     9             , b3iib2  , b2iib1  , b2iib2
*endif MHD
```

Only `v1` has three boundary values that can be set (at `i=is,ism1,ism2`). Since just the skin values of `emf2` and `emf3` can be set, there is no second or third boundary variable available. All remaining variables are zone-centred in the i-direction, and thus have two boundary values to set (at `i=ism1,ism2`).

B.9 OIB—Outer I Boundary control (NMLSTS)

This namelist specifies both the boundary type and the in-flow values of all the flow variables for the outer-i boundary. These variables are *not* declared if the *EDITOR* macro `ISYM` is set. See comments for `IIB`.

```
parameter                    description                              default

noib  (j,k)     "btype" of outer i boundary on sweep j,k                 2
goib            "gtype" of entire outer i boundary                       2
**oib1(j,k)     first  outer i boundary value of variable **          **floor
                for sweep j,k (flow in only)
**oib2(j,k)     second outer i boundary value of variable **          **floor
                for sweep j,k (flow in only)
**oib3(j,k)     third  outer i boundary value of variable **          **floor
                for sweep j,k (flow in only)
gpoib (j,k)     analytical or preset values for gp on oib               0.0
                for sweep j,k (goib=3 only)

       namelist / oib      /
     1               noib    , goib    , doib1   , doib2   , v1oib1
     2             , v1oib2  , v1oib3  , v2oib1  , v2oib2  , v3oib1
     3             , v3oib2
*if -def,ISO
     4             , e1oib1  , e1oib2
*endif -ISO
*if def,TWOFLUID
     5             , e2oib1  , e2oib2
*endif TWOFLUID
*if def,RADIATION
     6             , eroib1  , eroib2
*endif RADIATION
*if def,GRAV
     7             , gpoib
*endif GRAV
*if def,MHD
     8             , emf1oib1, emf1oib2, emf2oib1, emf3oib1, b3oib1
     9             , b3oib2  , b2oib1  , b2oib2
*endif MHD
```

B.10 IJB—Inner J Boundary control (NMLSTS)

This namelist specifies both the boundary type and the in-flow values of all the flow variables for the inner-j boundary. These variables are *not* declared if the *EDITOR* macro JSYM is set. See comments for IIB.

```
parameter                    description                              default

nijb  (k,i)     "btype" of inner j boundary on sweep k,i                 2
gijb            "gtype" of entire inner j boundary                       2
**ijb1(k,i)     first  inner j boundary value of variable **          **floor
                for sweep k,i (flow in only)
**ijb2(k,i)     second inner j boundary value of variable **          **floor
                for sweep k,i (flow in only)
**ijb3(k,i)     third  inner j boundary value of variable **          **floor
                for sweep k,i (flow in only)
gpijb (k,i)     analytical or preset values for gp on ijb               0.0

       namelist / ijb      /
     1               nijb    , gijb    , dijb1   , dijb2   , v1ijb1
     2             , v1ijb2  , v2ijb1  , v2ijb2  , v2ijb3  , v3ijb1
     3             , v3ijb2
*if -def,ISO
```

```
     4             , e1ijb1  , e1ijb2
*endif -ISO
*if def,TWOFLUID
     5             , e2ijb1  , e2ijb2
*endif TWOFLUID
*if def,RADIATION
     6             , erijb1  , erijb2
*endif RADIATION
*if def,GRAV
     7             , gpijb
*endif GRAV
*if def,MHD
     8             , emf2ijb1, emf2ijb2, emf3ijb1, emf1ijb1, b1ijb1
     9             , b1ijb2  , b3ijb1  , b3ijb2
*endif MHD
```

B.11 OJB—Outer J Boundary control (NMLSTS)

This namelist specifies both the boundary type and the in-flow values of all the flow variables for the outer-j boundary. These variables are *not* declared if the *EDITOR* macro JSYM is set. See comments for IIB.

```
parameter                    description                          default

nojb  (k,i)     "btype" of outer j boundary on sweep k,i             2
gojb            "gtype" of entire outer j boundary                   2
**ojb1(k,i)     first  outer j boundary value of variable **      **floor
                for sweep k,i (flow in only)
**ojb2(k,i)     second outer j boundary value of variable **      **floor
                for sweep k,i (flow in only)
**ojb3(k,i)     third  outer j boundary value of variable **      **floor
                for sweep k,i (flow in only)
gpojb (k,i)     analytical or preset values for gp on ojb           0.0
```

```
       namelist / ojb      /
     1              nojb    , gojb    , dojb1   , dojb2   , v1ojb1
     2             , v1ojb2  , v2ojb1  , v2ojb2  , v2ojb3  , v3ojb1
     3             , v3ojb2
*if -def,ISO
     4             , e1ojb1  , e1ojb2
*endif -ISO
*if def,TWOFLUID
     5             , e2ojb1  , e2ojb2
*endif TWOFLUID
*if def,RADIATION
     6             , erojb1  , erojb2
*endif RADIATION
*if def,GRAV
     7             , gpojb
*endif GRAV
*if def,MHD
     8             , emf2ojb1, emf2ojb2, emf3ojb1, emf1ojb1, b1ojb1
     9             , b1ojb2  , b3ojb1  , b3ojb2
*endif MHD
```

B.12 IKB—Inner K Boundary control (NMLSTS)

This namelist specifies both the boundary type and the in-flow values of all the flow variables for the inner-k boundary. These variables are *not* declared if the *EDITOR* macro KSYM is set. See comments for IIB.

parameter	description	default
nikb (i,j)	"btype" of inner k boundary on sweep i,j	2
gikb	"gtype" of entire inner k boundary	2
**ikb1(i,j)	first inner k boundary value of variable ** for sweep i,j (flow in only)	**floor
**ikb2(i,j)	second inner k boundary value of variable ** for sweep i,j (flow in only)	**floor
**ikb3(i,j)	third inner k boundary value of variable ** for sweep i,j (flow in only)	**floor
gpikb (i,j)	analytical or preset values for gp on ikb	0.0

```
       namelist / ikb      /
     1               nikb    , gikb    , dikb1   , dikb2   , v1ikb1
     2             , v1ikb2  , v2ikb1  , v2ikb2  , v3ikb1  , v3ikb2
     3             , v3ikb3
*if -def,ISO
     4             , e1ikb1  , e1ikb2
*endif -ISO
*if def,TWOFLUID
     5             , e2ikb1  , e2ikb2
*endif TWOFLUID
*if def,RADIATION
     6             , erikb1  , erikb2
*endif RADIATION
*if def,GRAV
     7             , gpikb
*endif GRAV
*if def,MHD
     8             , emf3ikb1, emf3ikb2, emf1ikb1, emf2ikb1, b2ikb1
     9             , b2ikb2  , b1ikb1  , b1ikb2
*endif MHD
```

B.13 OKB—Outer K Boundary control (NMLSTS)

This namelist specifies both the boundary type and the in-flow values of all the flow variables for the outer-k boundary. These variables are *not* declared if the *EDITOR* macro KSYM is set. See comments for IIB.

parameter	description	default
nokb (i,j)	"btype" of outer k boundary on sweep i,j	2
gokb	"gtype" of entire outer k boundary	2
**okb1(i,j)	first outer k boundary value of variable ** for sweep i,j (flow in only)	**floor
**okb2(i,j)	second outer k boundary value of variable ** for sweep i,j (flow in only)	**floor
**okb3(i,j)	third outer k boundary value of variable ** for sweep i,j (flow in only)	**floor
gpokb (i,j)	analytical or preset values for gp on okb	0.0

```
       namelist / okb      /
     1               nokb    , gokb    , dokb1   , dokb2   , v1okb1
     2             , v1okb2  , v2okb1  , v2okb2  , v3okb1  , v3okb2
     3             , v3okb3
*if -def,ISO
     4             , e1okb1  , e1okb2
*endif -ISO
*if def,TWOFLUID
     5             , e2okb1  , e2okb2
*endif TWOFLUID
*if def,RADIATION
     6             , erokb1  , erokb2
*endif RADIATION
*if def,GRAV
     7             , gpokb
*endif GRAV
*if def,MHD
     8             , emf3okb1, emf3okb2, emf1okb1, emf2okb1, b2okb1
     9             , b2okb2  , b1okb1  , b1okb2
*endif MHD
```

B.14 GRVCON—GRaVity CONtrol, (NMLSTS)

Gravitational self-potential is switched on by defining GRAV and aliasing GRAVITY to the desired gravity routine. If GRAVITY is aliased to gravity, the user must select the desired Poisson-solver by specifying a value for grvalg.

In addition, a point mass potential can be included by specifying a positive value for ptmass. Point mass potentials do not require defining GRAV, do not call the GRAVITY module, and are not included in the array gp. Their effect is explicitly added as velocity source terms in the routines stv1, stv2, and stv3. Thus, a point-mass potential may be used in conjunction with self-gravity or with self-gravity turned off.

parameter	description	default
gcnst	gravitational constant = 0.25/pi for unitless calculations = 6.67259d-11 for mks (known only to 6 sig.\ figs.) = 6.67259d-08 for cgs	0.25/pi
ptmass	fixed central point mass object. If using scaled units, ptmass (scaled point mass M) will depend on gcnst and the scaling for density and grid size. For gcnst = 1/4pi, ptmass = (4 pi G M) / (ds rs**3), where ds is the density scale and rs is the length scale. For M = 1 solar mass, ds = 3.0e5 hydrogen atoms per m**3, and rs = 1.0e3 AU, ptmass ~ 1.	0.0
iptmass	i index of point mass	ismn
jptmass	j index of point mass	jsmn
kptmass	k index of point mass	ksmn
grvalg	self-gravitational algorithm to be used. .le. 1 => Successive Overrelaxation (SOR) .eq. 2 => Full Multi-grid (FMG)	2
gcycle	maximum number of iterations for SOR number of V-cycles for FMG	GRAVITYITER
epsgrv	maximum tolerance for convergence of	GRAVITYERROR

```
               Poisson solvers
nrelax         a full multigrid parameter                              0

       namelist / grvcon   /
     1               gcnst   , ptmass  , iptmass , jptmass , kptmass
     2             , grvalg  , gcycle  , epsgrv  , nrelax
```

B.15 EQOS—EQuation Of State control (NMLSTS)

This namelist specifies the parameters which control the equation of state. Using all the defaults is recommended, unless a different adiabatic constant (gamma) is required. Note that if an isothermal equation of state is desired, setting the *EDITOR* definition `ISO` in addition to setting `niso` = 1 will allow execution to take advantage of the reduced computations necessary for isothermal systems. Parameters dimensioned with `nmat` allow for values to be set for both fluids if `TWOFLUID` is set, with the first element reserved for the first fluid (that which exists when `TWOFLUID` is not set), and the second element for the second (possibly diffusive) fluid enabled when `TWOFLUID` is set.

```
parameter                   description                             default

gamma (nmat)    ratio of specific heats                              5/3
rgas  (nmat)    gas constant                                          1.0
niso  (nmat)    =0 => adiabatic  eos                                  0
                =1 => isothermal eos
ciso  (nmat)    isothermal sound speed                                1.0
rmetal(nmat)    metallicity => cooling strength  M-MML                0.0
diffc1, diffc2  diffusion coefficient (for the second fluid) is       0.0
                set to diffc1 / B**diffc2

       namelist / eqos     /
     1               gamma   , rgas    , niso    , ciso    , rmetal
     2             , diffc1  , diffc2
```

B.16 GCON—Grid motion CONtrol (NMLSTS)

This namelist sets the parameters for grid motion, should a partial tracking of the flow be required. This feature has been dormant for years, and should this feature be desired, some code development may be required.

```
parameter                   description                             default

x1fac           x1 motion factor                                      0.0
                < 0 gives "Lagrangian" tracking in x1 lines
x2fac           x2 motion factor                                      0.0
                < 0 gives "Lagrangian" tracking in x2 lines
x3fac           x3 motion factor                                      0.0
                < 0 gives "Lagrangian" tracking in x3 lines
ia              i<ia => zone ratio is preserved in x1 lines         is=3
ja              j<ja => zone ratio is preserved in x2 lines         js=3
ka              k<ka => zone ratio is preserved in x3 lines         ks=3
igcon           selects grid treatment:                               0
                =0 => separate motion
                =1 => averaged motion
```

```
              =2 => tracking x1, x2, and x3 boundaries
              =3 => averaged boundary tracking
              =4 => input grid boundary speeds
                    vg1(ie) = x1fac * central sound speed
                    vg2(je) = x2fac * central sound speed
                    vg3(ke) = x3fac * central sound speed

      namelist / gcon     /
     1               x1fac   , x2fac   , x3fac   , ia      , ja
     2             , ka      , igcon
```

B.17 EXTCON—grid EXTension CONtrol (NMLSTS)

This namelist controls the grid extension feature of the code. This is useful only for problems in which a shock separates quiescent material (which does not require updating) from material requiring computations. As the shock propagates across the grid, more zones are added to the computational domain until the entire domain has been included. Because quiescent zones are not being updated, a substantial savings in computation time could be realised. Use this feature with caution. Improper use can be disastrous.

```
parameter                     description                          default

istretch(1) .le.  0  => perform computations over entire i-domain    0
            .gt.  0  => i-index of first zone in initial i-domain
istretch(2)             i-index of last  zone in initial i-domain.   0
            .le. (1) => istretch(2)=istretch(1)+istretch(3)-1
istretch(3) .le.  0  => 10                                           0
istretch(4) .le.  0  => istretch(3)                                  0
jstretch(1,2,3,4)       same as "istretch", but for 2-direction.
kstretch(1,2,3,4)       same as "istretch", but for 3-direction.
extvar                  specifies variable used to detect dis-       'd'
                        turbance in the quiescent ambient medium
                        (character*2).  Legal values are: 'd ',
                        'e ' (pressure), 'se' (temperature).
```

Note that `ismn` and `iemx` are the user-imposed limits of the grid in the `i`-direction, while `is` and `ie` are the `i`-limits of the do-loops. With grid extension off, `is = ismn` and `ie = iemx`. With grid extension on, `is .ge. ismn` and `ie .le. iemx` (§C.1). `is` is decremented by `istretch(3)` and/or `ie` is incremented by `istretch(4)` whenever the quiescent value of the specified variable (`extvar`) changes by 3% within 5 zones of the current domain boundary. Note that `is` will not be permitted to fall below `ismn` and `ie` will not be permitted to rise above `iemx`. Grid extension in the `i`-direction is turned off by keeping `istretch(1) = 0` (its default value).

An entirely analogous discussion holds for the `j`- and `k`-directions.

```
      namelist / extcon   /
     1               istretch, jstretch, kstretch, extvar
```

B.18 PLT1CON—PLoT (1-D) CONtrol (NMLSTS)

This namelist controls the 1-D graphics. During a run, as many as `nios` 1-D slices may be specified for each variable plotted, where `nios` is a parameter set before compilation (default

value for nios = 20). For every slice chosen, a file (in either metacode or postscript) is created with a plot generated for each variable specified. These plots may be arranged in the same frame or separate frames, and can have any rectangular shape desired. All plots are of publication quality. Each 1-D slice (bounded by x1p1mn, x1p1mx, *etc.*) runs parallel to one of the axes of the computational grid. To specify the slice uniquely, two of iplt1, jplt1, and kplt1 must be set.

For 1-D runs such as shock-tube tests, the analytical solution may be overlaid by setting *EDITOR* macro RIEMANN and setting the namelist parameters ip1soln and xdiscp1. The Riemann solver (courtesy of Tom Jones) takes the end points of the 1-D run as the left and right states, and thus the run should stop before the boundaries are reached.

N.B. For restarted runs in which the computation is resumed on a larger or smaller grid and where the default values for x1p1mn, x1p1mx, *etc.* were used in the initial run, it will be necessary to set x1p1mn, x1p1mx, *etc.* in the input deck for the restarted run to the extrema of the new grid if the plots are to extend to the bounds of the new grid. Otherwise, the plots will be bound by the old grid.

parameter	description	default
iplt1dir(nios)	axis parallel to slice. 0 => no plots 1, 2, 3 => 1-, 2-, 3-direction	0
iplt1 (nios)	i index of 1-D plot in 2- or 3-direction	(is+ie)/2
jplt1 (nios)	j index of 1-D plot in 3- or 1-direction	(js+je)/2
kplt1 (nios)	k index of 1-D plot in 1- or 2-direction	(ks+ke)/2
dtplt1	physical (problem) time interval between 1-D plot dumps. 0.0 => no plots.	0.0
nplt1dmp	the sequential number for the next 1-D plot file < 0 => nplt1dmp = iplt1dmp	-1
plt1var (niov)	names of variables to be plotted (character*2). Valid names are 'd ' (density), 'e1' (1st int. energy), 'e2' (2nd int. energy), 'u1' (1st (specific int. energy), 'u2' (2nd specific int. energy), 'et' (total energy density) 'p1' (1st thermal pressure), 'p2' (magnetic pressure), 'p3' (1st ther. + mag. pres.), 'p4' (2nd thermal pressure), 'p5' (1st ther. + 2nd ther. pres.), 'p6' (mag. + 2nd ther. pres.), 'p7' (1st ther. + mag. + 2nd ther. pres.), 'k1' (entropy of 1st fluid), 'k2' (2nd entropy), 'ka' (avg. 1st + 2nd ent.), 'v1', 'v2', 'v3' (velocity components), 'v' (speed), 'vv' [div(v)], 's1', 's2', 's3' (momentum components), 'w1', 'w2', 'w3' (vorticity components), 'w' (vorticity norm), 'm' (Mach number), 'ma' (Alfven Mach number), 'ms' (slow MS Mach number), 'mf' (fast MS Mach number), 'gp' (gravitational potential), 'pg' (pseudo-gravitational potential), 'a1', 'a2', 'a3', (vector potential components), 'a ' (vector potential norm), 'b1', 'b2', 'b3' (magnetic field components), 'bP', 'bR' (phi- and radial magnetic field components), 'b' (magnetic field norm), 'bd' (magnetic field norm/density), 'ps' [atan(b3/b2)], 'pa' [pitch angle; atan(b1/bP)], 'bt' (plasma beta), 'j1', 'j2', 'j3' (current density components), 'j ' (current density norm), 'br'	'zz'

```
                (bremsstrahlung emissivity), 'sy' (synchrotron
                emissivity), 'om' (angular velocity), 'l '
                (angular momentum), 'lt' (angular momentum radial
                transport), 'kt' (kinetic viscous radial trans.),
                'gl' [d(r lt) / dr], 'gk' [d(r kt) / dr], 'hy'
                [(1/rho) dp/dr - 3/(2r**2)], B3 (b3*b3).
nlplt1  (niov)  = 1 => linear-linear       plot                    1
                = 2 => log(y)-linear(x)    plot
                = 3 => linear(y)-log(x)    plot
                = 4 => log-log             plot
ip1mean (niov)  =0 => ordinary 1-D slices; no means taken.         0
                =1 => 1-D slices are filled with means of
                      variable across orthogonal plane
plt1min (niov)  minimum value to be plotted.                       0.0
plt1max (niov)  maximum value to be plotted.                       0.0
plt1ref (niov)  reference line to be drawn on plot (< -1.0e30 => no
                line, default)
ip1soln (niov)  = 1 => Analytical solution to Riemann problem using  0
                       end points of slice as left- and right-states
                       overlaid.
                = 0 => no overlay
xdiscp1 (nios)  location of discontinuity (default: middle of slice)
iplt1mm         = 0 => use input "plt1min", "plt1max" for each plot  1
                = 1 => compute "plt1min" and "plt1max" for plots
                       If "plt1min" and "plt1max" are 0.0 for a
                       particular variable, compute extrema for
                       that variable as though "iplt1mm" were 1
units           sets the units of angular dimensions (character*2)   'rd'
                'dg' => degrees, 'rd' => radians, 'pi' => units
                of pi radians
x1p1mn  (nios)  minimum x1 of slice                          x1a(is)
x1p1mx  (nios)  maximum x1 of slice                          x1a(ie+1)
x2p1mn  (nios)  minimum x2 of slice                          x2a(js)
x2p1mx  (nios)  maximum x2 of slice                          x2a(je+1)
x3p1mn  (nios)  minimum x3 of slice                          x3a(ks)
x3p1mx  (nios)  maximum x3 of slice                          x3a(ke+1)
ip1mn   (nios)  i-index of minimum x1 of slice                    0
ip1mx   (nios)  i-index of maximum x1 of slice                    0
jp1mn   (nios)  j-index of minimum x2 of slice                    0
jp1mx   (nios)  j-index of maximum x2 of slice                    0
kp1mn   (nios)  k-index of minimum x3 of slice                    0
kp1mx   (nios)  k-index of maximum x3 of slice                    0
corl            =1 => use open Circles, one per zone              2
                =2 => use Line segments to connect zone values
aspect          < 1.0 => plots are wide and short                 1.0
                = 1.0 => square plots
                > 1.0 => plots are tall and narrow
np1h            number of plots horizontally per frame            1
np1v            number of plots vertically   per frame            1
allxl           = 2 => all x labels/annotations are drawn         2
                = 1 => only the x-label/annotations on the bottom
                       row of plots on the frame are drawn.
                = 0 => labels aren't even drawn on the bottom row.
allyl           = 2 => all y labels/annotations are drawn         2
                = 1 => only the y-label/annotations on the left
                       column of plots on the frame are drawn.
                = 0 => labels aren't even drawn on the left column.
norpp1          = 1 => use NCAR graphics library for 1-D plots    2
                = 2 => use PSPLOT graphics library for 1-D plots
```

```
ip1hdr          = 0 => suppresses header                              1
                = 1 => header is printed on top    of plot
ip1ftr          = 0 => suppresses footer                              1
                = 1 => footer is printed on bottom of plot
p1xlab  (nios)  character string containing NCAR mark-up language
                for the desired x-label.  Default is the generic
                coordinate label (x1, x2, x3).
p1ylab  (niov)  character string containing NCAR mark-up language
                for the desired y-label.  Most defaults are fine,
                unless, for example, v_\phi is desired over v_3, etc.
porsp1          = 1 => heavy lines, for high resolution printer       2
                = 2 => light lines, suitable for CRT screen
```

Note that two of `iplt1`, `jplt1`, and `kplt1` must be specified for each slice. If `ip1mn`, *etc.* is 0, `x1p1mn`, *etc.* is used instead.

If you are using *NCAR* graphics (`norpp1=1`), you will need to link all *NCAR* graphics libraries to your executable (see your system administrator if you do not know what or where these libraries are) as well as two user-created libraries, `grfx03.a` and `psplot.a`. If you are using *PSPLOT* (`norpp1=2`), then you will need to link *either* `grfx03.a`, `psplot.a` plus all the *NCAR* graphics libraries as if you were using *NCAR*, *or* `grfx03.a`, `psplot.a`, `noncar.a`, and no *NCAR* graphics libraries.

```
  namelist / plt1con  /
1               iplt1dir, iplt1   , jplt1   , kplt1   , dtplt1
2             , plt1var , nlplt1  , ip1mean , plt1min , plt1max
3             , plt1ref , ip1soln , xdiscp1 , iplt1mm , units
4             , x1p1mn  , x1p1mx  , x2p1mn  , x2p1mx  , x3p1mn
5             , x3p1mx  , ip1mn   , ip1mx   , jp1mn   , jp1mx
6             , kp1mn   , kp1mx   , corl    , aspect  , np1h
7             , np1v    , allxl   , allyl   , norpp1  , ip1hdr
8             , ip1ftr  , p1xlab  , p1ylab  , porsp1  , nplt1dmp
```

B.19 PLT2CON—PLoT (2-D) CONtrol (NMLSTS)

This namelist controls the 2-D graphics. During a run, as many as `nios` 2-D slices may be specified for each variable plotted. For every slice chosen, a file (metacode or postscript) is created with a plot generated for each variable specified. The normal to each slice is parallel to one of the axes of the computational grid and is specified by `iplt2dir`. The extent of the slice is limited by `x1p2mn`, `x1p2mx`, *etc.*, while the index at the base of the normal to the slice is given by `lplt2`.

2-D graphics are in the form of contours (scalars and vector components normal to the image plane), vectors (poloidal vector components), or both for combined plots. Colour contours may also be specified if using the *PSPLOT* option. Plots are of publication quality and come fully labelled, including a time stamp for easy identification. Unlike the 1-D plots, only one plot may be written to each frame. However, the plot may be scaled down (`p2scale`) if desired.

N.B. For restarted runs in which the computation is resumed on a larger or smaller grid, and where the default values for `x1p2mn`, `x1p2mx`, *etc.* were used in the initial run, it will be necessary to set `x1p2mn`, `x1p2mx`, *etc.* in the input deck for the restarted run to the extrema of the new grid if the plots are to extend to the bounds of the new grid. Otherwise, the plots will be bound by the old grid.

parameter	description	default
iplt2dir(nios)	direction of normal to image plane. 0 => no plots; 1, 2, 3 => 1-, 2-, 3-direction	0
lplt2 (nios)	level of 2-D plot (value of 1-, 2-, or 3-index)	(is+ie)/2
iplt2avg(nios)	= 1 => averages slice with lplt2-1 = 0 => no average taken	0
dtplt2	physical (problem) time interval between 2-D plot dumps. 0.0 => no plots.	0.0
dtcntr	physical (problem) time interval between diagnostic contour dumps 0.0 => no plots.	0.0
dtvctr	physical (problem) time interval between diagnostic vector dumps 0.0 => no plots.	0.0
nplt2dmp	the sequential number for the next 2-D plot file < 0 => nplt2dmp = iplt2dmp	-1
plt2var (niov)	names of variables to be plotted (character*2). Valid names are 'd ', 'e1', 'e2', 'u1', 'u2', 'et', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'k1', 'k2', 'ka', 'v1', 'v2', 'v3', 'v ', 'vv', 'vp' (poloidal velocity), 'vn' (normal velocity), 'va' (pol. vect. + normal cont.), 'vd' (vel. vect. + density cont.), 's1', 's2', 's3', 'sp' (poloidal momentum), 'sn' (normal momentum), 'sa' (pol. vect. + normal cont.), 'sd' (momentum vect. + density cont.), 'w1', 'w2', 'w3', 'w ', 'wp' (poloidal vorticity), 'wn' (normal vort.), 'wa' (pol. vect. + normal cont.), 'm ', 'ma', 'ms', 'mf', 'gp', 'pg', 'a1', 'a2', 'a3', 'a ', 'ap' (poloidal vector pot.), 'an' (normal vector potential) 'aa' (pol. vect. + normal cont.), 'fn' (normal magnetic flux function), 'b1', 'b2', 'b3', 'bP', 'bR', 'b ', 'bt', 'bp' (poloidal magnetic field), 'bn' (normal magnetic field), 'ba' (pol. vect. + normal cont.), 'bd' (pol. magnetic field vect. + density cont.), 'pa', 'j1', 'j2', 'j3', 'j ', 'jp' (poloidal current density), 'jn' (normal current density), 'ja' (pol. vect. + normal cont.), 'dj' (pol. current vect. + density cont.).	'zz'
ngplt2 (niov)	= 1 => contours are spaced evenly > 1 => contours are spaced geometrically such that the first two contours are spaced "ngplt2" times closer than for evenly spaced contours. <-1 => contours are spaced geometrically such that the last two contours are spaced "-ngplt2" times closer than for evenly spaced contours.	1
vnorm (niov)	>=1.0 => normalise vectors, draw all vectors with original length > 1.0e-24 * vmax > 0.0 => do not normalise vectors, draw vectors >= < 1.0 vnorm * vmax only = 0.0 => do not normalise vectors, draw vectors >= 0.04 * vmax only (default) < 0.0 => normalise vectors, draw all vectors with original length > 10**vnorm * vmax	
plt2min (niov)	minimum value to be contoured.	0.0
plt2max (niov)	maximum value to be contoured.	0.0
iplt2mm	= 0 => use input "plt2min", "plt2max" for plots = 1 => compute "plt2min" and "plt2max" for plots	1

```
                        If "plt2min" and "plt2max" are 0.0, compute
                        them as if "iplt2mm" were 1
numcl    (niov)  =0 => min(20,ncls) evenly spaced contours are          0
                       drawn between "plt2min" and "plt2max"
                 >0 => number of evenly spaced contours to draw
                       between "plt2min" and "plt2max" (max ncls).
                 <0 => abs(numcl) contours drawn and are specified
                       by "plt2min" and "levs".
                 if numcl(2:niov) are not set by user, they are set
                 to numcl(1).
levs     (ncls)  real array specifying multiples of "plt2min" to use
                 for contour levels (for numcl < 0 only).  Note that
                 the same multiples are used for all variables.
                 for contour levels (for numcl < 0 only).
dash             = 1 => -ve contours are drawn with dashed lines       0
                 = 0 => -ve contours are drawn with solid lines
                        and indistinguishable from positive contours.
hilo             = 1 => highs and lows are labelled.                   0
                 = 0 => highs and lows are not labelled (NCAR only).
vscale           scaling factor for vectors                          0.8
p2scale          .lt. 1.0d0 => scaling factor to reduce plot size     1.0
                 .ge. 1.0d0 => full size
dxvec    (nios)  x-increment between vectors (grid units)            1.0
dyvec    (nios)  y-increment between vectors (grid units)            1.0
units            sets the angular units (character*2)                'rd'
                 'rd' => radians, 'pi' => units of pi radians
                 'dg' => degrees
x1p2mn   (nios)  minimum x1 of plot window                        x1a(is)
x1p2mx   (nios)  maximum x1 of plot window                        x1a(ie+1)
x2p2mn   (nios)  minimum x2 of plot window                        x2a(js)
x2p2mx   (nios)  maximum x2 of plot window                        x2a(je+1)
x3p2mn   (nios)  minimum x3 of plot window                        x3a(ks)
x3p2mx   (nios)  maximum x3 of plot window                        x3a(ke+1)
ip2mn    (nios)  i-index of minimum x1 of plot window                   0
ip2mx    (nios)  i-index of maximum x1 of plot window                   0
jp2mn    (nios)  j-index of minimum x2 of plot window                   0
jp2mx    (nios)  j-index of maximum x2 of plot window                   0
kp2mn    (nios)  k-index of minimum x3 of plot window                   0
kp2mx    (nios)  k-index of maximum x3 of plot window                   0
iflipp2  (nios)  = 0 => no flipping of plots                            0
                 = 1 => plot is flipped about x-axis before writing
jflipp2  (nios)  = 0 => no flipping of plots                            0
                 = 1 => plot is flipped about y-axis before writing
ireflp2  (nios)  = 0 => plots are not reflected about x-axis            0
                 = 1 => plot is reflected about x-axis before writing
                        rendering the plot twice as tall
jreflp2  (nios)  = 0 => plots are not reflected about y-axis            0
                 = 1 => plot is reflected about y-axis before writing
                       rendering the plot twice as long
porsp2           = 1 => heavy lines, suitable for high resolution       2
                        printer.
                 = 2 => light lines, suitable for CRT screen.
ip2ftr           = 0 => suppresses footer                               1
                 = 1 => footer is printed on bottom of plot
ip2hdr           = 0 => suppresses header                               1
                 = 1 => header is printed on top of plot
ip2xlab          = 0 => suppresses xlabel (but not x-annotations)       1
                 = 1 => xlabel is put below horizontal axis
ip2ylab          = 0 => suppresses ylabel (but not y-annotations)       1
```

```
                = 1 => ylabel is put beside vertical  axis
p2xlab  (nios)  character string containing NCAR mark-up language
p2ylab  (nios)  for the desired x- and y-labels.  Default is the
                generic coordinate label (x1, x2, x3).
norpp2          = 1 => use NCAR graphics library for 2-D plots        2
                = 2 => use PSPLOT graphics library for 2-D plots
pscolr  (niov)  = 0 => do not use colour or grey-scale                0
                = 1 => use grey-scale to fill between contour
                = 2 => use colour     to fill between contour (PSPLOT)
                if pscolr(2:niov) are not set by user, they are set
                to pscolr(1).
pscntr  (niov)  = 0 => do not overlay contours on colours             0
                = 1 => overlay colours with thin contours (PSPLOT)
                if pscntr(2:niov) are not set by user, they are set
                to pscntr(1).
psgrid          = 0 => do not overlay grid lines onto plots           0
                > 0            => overlay a-grid
                < 0 or > 10000 => overlay b grid
                every mod(|psgrid|,10000)th grid line drawn (PSPLOT)
```

If `ip2mn`, *etc.* is 0, `x1p2mn`, *etc.* is used instead.

When using *PSPLOT*, setting `pscolr=0` will give contour plots with contour levels listed in the footer, just as for *NCAR* plots. For `pscolr=1`, contours are only included if `pscntr=1`, and a vertical colour bar to the right of the plot replaces the contour levels listed in the footer.

If you are using *NCAR* graphics (`norpp2=1`), you will need to link all *NCAR* graphics libraries to your executable (see your system administrator if you do not know what or where these libraries are) as well as two user-created libraries, `grfx03.a` and `psplot.a`. If you are using *PSPLOT* (`norpp2=2`), then you will need to link *either* `grfx03.a`, `psplot.a` plus all the *NCAR* graphics libraries as if you were using *NCAR*, *or* `grfx03.a`, `psplot.a`, `noncar.a`, and no *NCAR* graphics libraries.

```
      namelist / plt2con  /
     1              iplt2dir, lplt2   , iplt2avg, dtplt2  , dtcntr
     2            , dtvctr  , nplt2dmp, plt2var , ngplt2  , vnorm
     3            , plt2min , plt2max , iplt2mm , numcl   , levs
     4            , dash    , hilo    , vscale  , p2scale , dxvec
     5            , dyvec   , units   , x1p2mn  , x1p2mx  , x2p2mn
     6            , x2p2mx  , x3p2mn  , x3p2mx  , ip2mn   , ip2mx
     7            , jp2mn   , jp2mx   , kp2mn   , kp2mx   , iflipp2
     8            , jflipp2 , ireflp2 , jreflp2 , porsp2  , ip2ftr
     9            , ip2hdr  , ip2xlab , ip2ylab , p2xlab  , p2ylab
     1            , norpp2  , pscolr  , pscntr  , psgrid
```

B.20 PIXCON—PIXel graphics CONtrol (NMLSTS)

This namelist controls the pixel dumps. Pixel dumps are 2-D raster images of slices through the data volume, and are rebinned to a uniform, square Cartesian grid. During a run, as many as `nios` slices may be specified for each variable plotted. A single pixel dump is created for every variable and every slice specified. The extent of the pixel slice can be limited by setting `x1pxmn`, `x1pxmx`, *etc.* The normal to the pixel slice is parallel to one of the axes of the computational grid and is specified by `ipixdir`. The index at the base of the normal is given by `lpix`.

Pixel dumps are designed to provide a format for generating smooth qualitative temporal animations of the flow variables. Aim for about 500 dumps for each animation. They may be written in either raw format (`rorhpix=1`, one byte per datum) or *HDF* (`rorhpix=2`, four bytes per datum).

Raw format files are small, and so numerous images may be generated with a relatively small amount of disc space. However, the low dynamic range of the images (256) dictates that the data be bracketed and perhaps even dumped logarithmically in order to render the salient features visible. The data may be bracketed automatically (`ipixmm=1`), in which case differences from one image to the next will be caused by both the evolution of the flow *and* the fluctuations of the extrema which are used to bracket the data. Alternatively, one may bracket the data manually (`ipixmm=0`) by setting values for `pixmin` and `pixmax`. This can be done by running the simulation until 10 to 20 pixel dumps have been generated for each variable with `ipixmm` set to 1. The extrema used to bracket the data are reported in the log file `zl`*nnn*`id`, and these can be used to set the extrema `pixmin` and `pixmax`. Now run the job from the beginning with `ipixmm` set to 0. If dumping the logarithm of a variable is desired, some experimentation may be needed in order to find the optimal value of `nlpix` (the dynamic range). However, the default value of 100 should be fine for most applications. Basically, the higher the absolute value for a positive (negative) `nlpix`, the more concentrated the colours will be at the low (high) end. Note that because of how the logarithm is taken, a variable need not be positive definite to use this feature.

HDF files are four times as big, and thus may cause disc and storage problems. However, because these images are four bytes deep, bracketing and converting to log are not necessary. In fact, these files may be used quantitatively as well as qualitatively. For *HDF*, the parameters `ncpix`, `ipixmm`, `pixmin`, `pixmax`, and `nlpix` are all ignored.

Polar slices are binned to a Cartesian grid before they are written to disc. If a polar grid includes very small zones near the origin, it may be best to request two pixel slices for each slice to be visualised. One slice would include the entire grid and mimic the resolution near the mid-radial regions (*i.e.*, oversample the outer grid, but undersample the inner grid). The second slice would include only the inner radial regions and would mimic the resolution of the inner grid.

The parameters which set the dimensions of the arrays for the pixel plots `(nxpx, nypx)` are independent of the parameters which set the dimensions of the flow variables `(in,jn,kn)`. Thus, in the case of a non-uniform grid, pixel dumps may be written with enough pixels to preserve the highest resolution on the grid.

N.B. For restarted runs in which the computation is resumed on a larger or smaller grid, and where the default values for `x1pxmn`, `x1pxmx`, *etc.* were used in the initial run, it will be necessary to set `x1pxmn`, `x1pxmx`, *etc.* in the input deck for the restarted run to the extrema of the new grid if the dumps are to extend to the bounds of the new grid. Otherwise, the dumps will be bound by the old grid.

parameter	description	default
ipixdir(nios)	direction of normal to image plane. 0 => no dumps; 1, 2, 3 => 1-, 2-, 3-direction	0
lpix (nios)	level of 2-D pixel dump (value of 1-, 2-, or 3-index)	(is+ie)/2
dtpix	problem time interval between pixel dumps	0.0

```
                0.0 => no pixel dumps
npixdmp         the sequential number for the next pixel file       -1
                <0 => npixdmp = ipixdmp
ncpix           number of colour contour levels in image           253
iref            =0 => no reflection                                  0
                =1 => q reflected across x-axis on output,
                      generates twice the y-pixels requested
jref            =0 => no reflection                                  0
                =1 => q reflected across y-axis on output,
                      generates twice the x-pixels requested
npi    (nios)   number of x-pixels in image slice                   nxpx
npj    (nios)   number of y-pixels in image slice                   nypx
pixvar (niov)   names of variables to be plotted (character*2).     'zz'
                Valid names are:  'd ', 'e1', 'e2', 'u1', 'u2',
                'et', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7',
                'k1', 'k2', 'ka', 'v1', 'v2', 'v3', 'vp', 'vn',
                'v ', 'vv', 's1', 's2', 's3', 'sp', 'sn', 'w1',
                'w2', 'w3', 'wp', 'wn', 'w ', 'm ', 'ma', 'ms',
                'mf', 'gp', 'pg', 'a1', 'a2', 'a3', 'ap', 'an',
                'a ', 'fn', 'b1', 'b2', 'b3', 'bP', 'bR', 'bp',
                'bn', 'b ', 'bt', 'pa', 'j1', 'j2', 'j3', 'jp',
                'jn', 'j ', 'sd' (skew-density)
nlpix  (niov)   =0 => store data                                     0
                >0 => store log10(data), concentrating colours at
                      low end.  Dynamic range = nlpix, 1 => 100.
                <0 => store log10(data), concentrating colours at
                      high end.  Dynamic range =-nlpix, -1 => -100.
pixmin (niov)   value of data to be assigned the minimum colour.     0.0
pixmax (niov)   value of data to be assigned the maximum colour.     0.0
ipixmm          =1 => compute "pixmin" and "pixmax" for images       1
                =0 => use input "pixmin", "pixmax" for images
                      If "pixmin" and "pixmax" are 0, compute
                      them as if "ipixmm" were 1
rorhpix         =1 => raw format used for dumps                      1
                =2 => HDF used for dumps (in which case, "nlpix",
                      "pixmin", and "pixmax" are ignored)
units           sets the angular units (character*2)                'rd'
                'rd' => radians, 'pi' => units of pi radians
                'dg' => degrees
x1pxmn (nios)   minimum x1 for pixel image                      x1a(is)
x1pxmx (nios)   maximum x1 for pixel image                      x1a(ie+1)
x2pxmn (nios)   minimum x2 for pixel image                      x2a(js)
x2pxmx (nios)   maximum x2 for pixel image                      x2a(je+1)
x3pxmn (nios)   minimum x3 for pixel image                      x3a(ks)
x3pxmx (nios)   maximum x3 for pixel image                      x3a(ke+1)
iflippx(nios)   =0 => no flipping of images                          0
                =1 => image is flipped about x-axis before writing
                      (size of image not changed, just flipped)
jflippx(nios)   =0 => no flipping of images                          0
                =1 => image is flipped about y-axis before writing

      namelist / pixcon   /
     1              ipixdir , lpix    , dtpix   , npixdmp , ncpix
     2            , iref    , jref    , npi     , npj     , pixvar
     3            , nlpix   , pixmin  , pixmax  , ipixmm  , rorhpix
     4            , units   , x1pxmn  , x1pxmx  , x2pxmn  , x2pxmx
     5            , x3pxmn  , x3pxmx  , iflippx , jflippx
```

B.21 VOXCON—VOXel graphics CONtrol (NMLSTS)

This namelist controls the voxel dumps of the 3-D data volume. These are the 3-D analogues of the 2-D pixel dumps, and are snapshots of the entire data volume. See comments in namelist pixcon above for discussion on raw format *vs. HDF*, bracketing, and dumping files logarithmically.

Voxel dumps are currently available for Cartesian (XYZ) and cylindrical (ZRP) geometries only. Cylindrical data are rebinned to a Cartesian grid before they are written to disc (similar to polar pixel dumps). The dimensions of the voxel dumps are limited by the parameters in, jn, and kn. In particular, the voxel dump may be no larger than in-1 × 2*jn-1 × 2*kn-1. For a uniform Cartesian grid, there is no reason to specify a voxel dump larger than the flow variable array. However, for non-uniform gridding in either or both of the 2- and 3-directions in XYZ coordinates, or in ZRP coordinates in general, the factor of 2 in both the j- and k-dimensions will allow the voxel dumps to represent better the regions in the computational grid with the highest resolution. 250 voxel dumps with four million voxels (from a one million zone computation) will require 1 Gbyte of disc space.

N.B. For restarted runs in which the computation is resumed on a larger or smaller grid, and where the default values for x1vxmn, x1vxmx, *etc.* were used in the initial run, it will be necessary to set x1vxmn, x1vxmx, *etc.* in the input deck for the restarted run to the extrema of the new grid if the dumps are to extend to the bounds of the new grid. Otherwise, the dumps will be bound by the old grid.

N.B. This feature has been dormant for at least a decade, and should be used with caution.

parameter	description	default
dtvox	problem time interval between voxel dumps 0.0 => no voxel dumps.	0.0
nvoxdmp	the sequential number for the next voxel file <0 => nvoxdmp = ivoxdmp	-1
ncvox	number of colour contour levels in image	253
nvi	number of voxels in 1-direction (.le. in-1) =0 => in-1	0
nvj	number of voxels in 2-direction (.le. 2*jn-1) =0 => increment in 2-dir. same as 1-dir.	0
nvk	number of voxels in 3-direction (.le. 2*kn-1) =0 => increment in 3-dir. same as 1-dir.	0
voxvar (niov)	names of variables to be plotted (character*2). Valid names are: 'd ', 'e1', 'e2', 'u1', 'u2', 'et', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'k1', 'k2', 'kt', 'v1', 'v2', 'v3', 'v ', 'vv', 's1', 's2', 's3', 'w1', 'w2', 'w3', 'w ', 'm ', 'ma',' mf', 'gp', 'pg', 'b1', 'b2', 'b3', 'b ', 'j1', 'j2', 'j3', 'j '	'zz'
nlvox (niov)	=0 => store data >0 => store log10(data), concentrating colours at low end. Dynamic range = nlvox, 1 => 100. <0 => store log10(data), concentrating colours at high end. Dynamic range =-nlvox, -1 => -100.	0
voxmin (niov)	value of data to be assigned the minimum colour.	0.0
voxmax (niov)	value of data to be assigned the maximum colour.	0.0
ivoxmm	=1 => compute "voxmin" and "voxmax" for images	1

```
                =0 => use input "voxmin", "voxmax" for images
                      If "voxmin" and "voxmax" are 0, compute
                      them as if "ivoxmm" were 1
rorhvox         =1 => raw format used for dumps                          1
                =2 => HDF used for dumps (in which case, "nlvox",
                      "voxmin", and "voxmax" are ignored)
units           sets the angular units (character*2)                    'rd'
                'rd' => radians, 'pi' => units of pi radians
                'dg' => degrees
x1vxmn          minimum x1 for voxel image                          x1a(is)
x1vxmx          maximum x1 for voxel image                          x1a(ie+1)
x2vxmn          minimum x2 for voxel image                          x2a(js)
x2vxmx          maximum x2 for voxel image                          x2a(je+1)
x3vxmn          minimum x3 for voxel image                          x3a(ks)
x3vxmx          maximum x3 for voxel image                          x3a(ke+1)

       namelist / voxcon   /
     1               dtvox   , nvoxdmp , ncvox   , nvi     , nvj
     2             , nvk     , voxvar  , nlvox   , voxmin  , voxmax
     3             , ivoxmm  , rorhvox , units   , x1vxmn  , x1vxmx
     4             , x2vxmn  , x2vxmx  , x3vxmn  , x3vxmx
```

B.22 USRCON—USeR dump CONtrol (NMLSTS)

This namelist is reserved for a user-supplied I/O subroutine aliased to USERDUMP (see App. A to see where USERDUMP is called). Additional namelist parameters may be added as needed.

```
parameter                     description                           default

dtusr     physical (problem) time interval between user dumps.      0.0
          0.0 => no user dumps
nusrdmp   the sequential number for the next user dump file        -1
          < 0 => nusrdmp = iusrdmp

       namelist / usrcon   /
     1               dtusr   , nusrdmp
```

B.23 HDFCON—HDF dump CONtrol (NMLSTS)

This namelist controls the *HDF* (Hierarchical Data Format) dumps. *HDF* is a format for data files developed at the NCSA. This format is fairly widely used, appearing in various commercial applications such as *IDL*. *HDF* dumps are 4 bytes deep, and contain the grid coordinates along with other useful information about the data.

In order to use *HDF*, it is necessary to link all the *HDF* libraries to your executable. If you don't know what or where these libraries are on your system, ask your system administrator who may have to download the (free) *HDF* libraries from the NCSA website (www.ncsa.uiuc.edu).

```
parameter                     description                           default

dthdf           physical (problem) time interval between hdf          0.0
                dumps.  0.0 => no hdf dumps
nhdfdmp         the sequential number for the next HDF file          -1
                <0 => nhdfdmp = ihdfdmp
```

```
hdfvar(niov)    names of variables to be dumped (character*2).       'zz'
                Valid names are 'to' ("total" dump => v1, v2,
                v3, b1, b2, b3, d, e1, e2, gp and pg all in the
                same file), 'd ', 'e1', 'e2', 'u1', 'u2', 'et',
                'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'k1',
                'k2', 'ka', 'v1', 'v2', 'v3', 'v ', 'vv', 's1',
                's2', 's3', 'w1', 'w2', 'w3', 'w ', 'm ', 'ma',
                'ms', 'mf', 'gp', 'pg', 'b1', 'b2', 'b3', 'b ',
                'j1', 'j2', 'j3', 'j '

      namelist / hdfcon   /
     1               dthdf   , nhdfdmp , hdfvar
```

B.24 TSLCON—Time SLice (history) dump CONtrol (NMLSTS)

This namelist controls the time slice data dumps. Various scalars, such as total mass, angular momenta, energy, extrema of variables, *etc.* are periodically written to an ascii file and/or a plot (*NCAR* or *PSPLOT* graphics). See §B.18 for what libraries are needed for *NCAR* and *PSPLOT* graphics respectively.

```
parameter                       description                          default

dttsl     physical (problem) time interval between time slice        0.0
          ascii dumps. 0.0 => no ascii time slices
ntsldmp   the sequential number for the next time slice file        -1
          <0 => ntsldmp = itsldmp
dttslp    physical (problem) time interval between time slice        0.0
          plot dumps. 0.0 => no metacode time slices
ntslpdmp  the sequential number for the next time slice plot file   -1
          <0 => ntslpdmp = itslpdmp
tslpmn    problem time for beginning of plot                         0.0
tslpmx    problem time for end of plot (0.0 => maximum time)         0.0
itslmn    minimum i-index of integration domain                     ismn
itslmx    maximum i-index of integration domain                     iemx
jtslmn    minimum j-index of integration domain                     jsmn
jtslmx    maximum j-index of integration domain                     jemx
ktslmn    minimum k-index of integration domain                     ksmn
ktslmx    maximum k-index of integration domain                     kemx
itslphdr  = 1 => write headers to time slice plot file               1
          = 0 => suppresses headers
itslpftr  = 1 => write footers to time slice plot file               1
          = 0 => suppresses footers
norptsl   = 1 => use NCAR graphics library for time slice plots      2
          = 2 => use PSPLOT graphics library for time slice plots
```

```
      namelist / tslcon   /
     1               dttsl   , ntsldmp , dttslp  , ntslpdmp, tslpmn
     2             , tslpmx  , itslmn  , itslmx  , jtslmn  , jtslmx
     3             , ktslmn  , ktslmx  , itslphdr, itslpftr, norptsl
```

B.25 DISCON—DISplay dump CONtrol (NMLSTS)

This namelist controls the display dumps of 2-D slices. During a run, as many as `nios` slices may be specified for each variable displayed. All display dumps generated during a run are dumped to the same ascii data file. The extent of the display slice can be limited by setting

idismn, idismx, *etc.* The normal to the display slice is parallel to one of the axes of the computational grid and is specified by idisdir. The index at the base of the normal is given by ldis.

The display format allows the user to view a small portion of the data quantitatively in a matrix format. The maximum amount of data that can be visualised at once from each specified variable and slice is 38 by 38. The data are scaled and converted to integers with a dynamic range anywhere from 100 to 10^6, depending on the amount of data being displayed. The data are arranged in a 2-D matrix and labelled with the grid indices and the scaling factor used to scale the data. (The functionality is similar to that of the task PRTIM in *AIPS*.)

N.B. For restarted runs in which the computation is resumed on a larger or smaller grid, and where the default values for idismn, idismx, *etc.* were used in the initial run, it will be necessary to set idismn, idismx, *etc.* in the input deck for the restarted run to the extrema of the new grid if the dumps are to extend to the bounds of the new grid. Otherwise, the dumps will be bound by the old grid.

parameter	description	default
idisdir(nios)	direction of normal to display slice: 0 => no dumps; 1, 2, 3 => 1-, 2-, 3-direction	0
ldis (nios)	level of 2-D display (value of 1-, 2-, or 3-index)	(is+ie)/2
dtdis	physical (problem) time interval between display dumps. 0.0 => no display dumps.	0.0
ndisdmp	the sequential number for the next display file <0 => ndisdmp = idisdmp	-1
disvar (niov)	names of variables to be displayed (character*2). Valid names are: 'd ', 'e1', 'e2', 'u1', 'u2', 'et', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'k1', 'k2', 'ka', 'v1', 'v2', 'v3', 'v ', 'vv', 's1', 's2', 's3', 'w1', 'w2', 'w3', 'w ', 'm ', 'ma', 'ms', 'mf', 'gp', 'pg', 'b1', 'b2', 'b3', 'b ', 'j1', 'j2', 'j3', 'j '	'zz'
idismn (nios)	bottom i-index of display window	is
idismx (nios)	top i-index of display window	ie
jdismn (nios)	bottom j-index of display window	js
jdismx (nios)	top j-index of display window	je
kdismn (nios)	bottom k-index of display window	ks
kdismx (nios)	top k-index of display window	ke

```
      namelist / discon   /
     1               idisdir , ldis    , dtdis   , ndisdmp , disvar
     2             , idismn  , idismx  , jdismn  , jdismx  , kdismn
     3             , kdismx
```

B.26 RADCON—RADio dump CONtrol (NMLSTS)

This namelist controls the *RADIO* dumps, which are 2-D pixel dumps of quantities integrated along the lines of sight through the data volume at arbitrary viewing angles (theta and phi). The volume integrated can be limited by setting x1rdmn, x1rdmx, *etc.* *RADIO* dumps are currently available for Cartesian (XYZ) and cylindrical (ZRP) geometries, with the latter not

fully debugged. See discussion in §B.20 regarding raw format *vs. HDF*, bracketing images, and dumping images logarithmically.

There are two types of integrated quantities: flow variables and emissivities. Many of the parameters listed below are for controlling the latter. For example, the Stokes parameters once integrated can be convolved with a beam, polarisation vectors may be plotted directly (rather than raster images), polarisation vectors may be superposed on total intensity raster images, and so on.

The "masks" (`*lower`, `*upper`, `dmask*`, and `bmask`) are useful in limiting which portion of the grid is included in the integration of the non-emissivity scalars. For example, if there is a contact discontinuity (CD) enclosing the region of interest, then there will be a jump in the density (`d`) along this interface. Thus, if `d`, for example, jumps from about 0.1 to about 1.0 across the CD, setting `dmask*=1.0`, and `dupper=0.5` would allow only the low density region (be it interior or exterior to the CD) to contribute to the line-of-sight integration of variable `*`. Alternatively, if the magnetic field is found only in the material of interest, setting `bmask*=1.0` would allow only material with magnetic field to be included in the integration of variable `*`. Finally, the variables `*lower` and `*upper` allow each variable to be masked by its own distribution. These can be set in addition to the density and/or magnetic field masks (`dmask*`, `bmask*`). For example, if only the compressive portions of the flow are to be integrated, then setting `xupper=0.0` will mean that only negative values of $\nabla \cdot \mathbf{v}$ will be included in the integration. All values excluded by the various masks will be given zero weight. In all cases, the default is no mask.

Reversing the palette (`nlrad<0`) is useful for raster images in which `radmin<0` and `radmax<0` (*e.g.*, negative velocity divergences). In these cases, it may be desirable to have the "maximum" colour correspond to the minimum pixel value (which has the greatest absolute value).

Note that the parameters which set the dimensions of the arrays for the *RADIO* pixel plots `(nxrd,nyrd)` are independent of the parameters which set the dimensions of the flow variables `(in,jn,kn)` and of the regular pixel slices `(nxpx,nypx)`.

N.B. For restarted runs in which the computation is resumed on a larger or smaller grid, and where the default values for `x1rdmn`, `x1rdmx`, *etc.* were used in the initial run, it will be necessary to set `x1rdmn`, `x1rdmx`, *etc.* in the input deck for the restarted run to the extrema of the new grid if the dumps are to extend to the bounds of the new grid. Otherwise, the dumps will be bound by the old grid.

parameter	description	default
dtrad	problem time interval between RADIO dumps 0.0 => no RADIO dumps.	0.0
nraddmp	the sequential number for the next radio file <0 => nraddmp = iraddmp	-1
thetamin	minimum angle between x1-axis and plane of sky	0.0
thetamax	maximum angle between x1-axis and plane of sky	0.0
dtheta	desired increment in "theta" between successive dumps	0.0
ntheta	number of values for "theta" between specified limits (overrides choice for "dtheta")	1
phimin	minimum azimuthal angle for lines of sight.	0.0
phimax	maximum azimuthal angle for lines of sight.	0.0
dphi	desired increment in "phi" between successive	0.0

```
                dumps
nphi            number of values for "phi" between specified          1
                limits (overrides choice for "dphi")
itype           0 => emissivities are not computed.                   2
                1 => Smith et al. emissivity (p**2)
                2 => CNB emissivity (function of d, p, B)
alpha           spectral index (itype=2 only)                         0.5
freq            frequency of RADIO "observation" (Hz).                5.0e9
brn0            fiducial number density for bremsstrahlung (m**-3)    1.0e6
brt0            fiducial temperature for bremsstrahlung (K)           1.0e6
brnu1,brnu2     limits of frequency band (Hz)                   1.000000e17
                                                                1.000001e17
ncrad           number of colour contour levels in images            253
radvar (niov)   names of variables to be plotted (character*2).       'zz'
                Currently, valid names are , 'A ' (pol'n position
                angle), 'AV' (pol'n position angle with pol'n
                vectors superposed), 'F ' (P/I), 'FV' (P/I with
                pol'n vectors superposed), 'I ' (total intensity),
                'IV' (total intensity with pol'n vectors
                superposed), 'P ' (pol'd intensity), 'PV' (pol'd
                intensity with pol'n vectors superposed), 'V '
                (pol'n vectors, black on white), 'VR' (pol'n
                vectors, white on black), 'D ' (density), 'E1'
                (first internal energy), 'U1' (first specific
                internal energy), 'B' (magnetic field strength),
                'SH' (velocity shear), 'VV' (velocity divergence),
                'BR' (bremsstrahlung), 'W ' (vorticity), 'M '
                (Mach Number), 'MA' (Alfven Mach number), 'MS'
                (slow MS Mach number), 'MF' (fast MS Mach number).
*lower          variable * is integrated along los provided it      -huge
                is greater than "*lower", where * is any one of
                d, e, se, pb, sh, vv, and br.
*upper          variable * is integrated along los provided it       huge
                is less than "*upper".
dmask*          density mask toggle for variable * (except d)        -1.0
                = 1.0 => "dlower" and "dupper" set int. limits
                = 0.0 => not
                =-1.0 => use "dmask"
dmask           density mask toggle for all variables                 0.0
                If "dmask*" .ne. -1.0, value of "dmask*" overrides
                "dmask" for variable * only
bmask*          B-field mask toggle for variable *                   -1.0
                = 1.0 => B-field extent sets int. limits
                = 0.0 => not
                =-1.0 => use "bmask"
bmask           B-field mask toggle for all variables                 0.0
                If "bmask*" .ne. -1.0, value of "bmask*" overrides
                "bmask" for variable * only
nlrad  (niov)   =0 => store data                                      0
                >0 => store log10(data), concentrating colours at
                      low end.  Dynamic range = nlrad, 1 => 100.
                <0 => store log10(data), concentrating colours at
                      high end.  Dynamic range =-nlrad, -1 => -100.
radmin (niov)   value of data to be assigned the minimum colour.      0.0
radmax (niov)   value of data to be assigned the maximum colour.      0.0
iradmm          =1 => compute "radmin" and "radmax" for images        1
                =0 => use input "radmin", "radmax" for images
                      If "radmin" and "radmax" are 0, compute
                      them as if "iradmm" were 1
```

```
rorhrad         =1 => raw format used for dumps                          1
                =2 => HDF used for dumps (in which case, "nlrad",
                      "radmin", and "radmax" are ignored)
icnvlv          0 => do not apply convolution                            0
                1 => apply convolution to Stokes parameters.
bmajor          major axis of convolving beam.                           1.0
bminor          minor axis of convolving beam.                           1.0
bpa             beam position angle (radians) measured counter-          0.0
                clockwise between major axis and +x axis.
cpb             "cells" per beam.                                        5.0
eorb            1 => E-vectors                                           2
                2 => B-vectors
porf            1 => vector length proportional to poli                  2
                2 => vector length proportional to fpol
bworb           1 => black and white pixel vectors                       1
                2 => black pixel vectors only
vlmin           vectors with length < vlmin*(max vector) not             0.001
                plotted.
icut            vectors are not plotted if total intensity               0.001
                toti < icut*max(toti)
pcut            vectors are not plotted if polarised intensity           0.001
                poli < pcut*max(poli)
pscale          scaling factor for polarisation vectors                  0.8
incpx           plot every incpx(th) vector in x-direction               1
incpy           plot every incpy(th) vector in y-direction               1
units           sets the angular units (character*2)                     'rd'
                'rd' => radians, 'pi' => units of pi radians
                'dg' => degrees
x1rdmn          minimum x1 for RADIO integration                  x1a(is)
x1rdmx          maximum x1 for RADIO integration                  x1a(ie+1)
x2rdmn          minimum x2 for RADIO integration                  x2a(js)
x2rdmx          maximum x2 for RADIO integration                  x2a(je+1)
x3rdmn          minimum x3 for RADIO integration                  x3a(ks)
x3rdmx          maximum x3 for RADIO integration                  x3a(ke+1)
iradbox         =0 => no box drawn around region of integration          0
                >0 => box drawn with highest colour
                <0 => box drawn with lowest  colour
```

```
      namelist / radcon   /
     1              dtrad   , nraddmp , thetamin, thetamax, ntheta
     2            , dtheta  , phimin  , phimax  , nphi    , dphi
     3            , itype   , alpha   , freq    , brn0    , brt0
     4            , brnu1   , brnu2   , ncrad   , radvar  , dlower
     5            , elower  , selower , blower  , shlower , vvlower
     6            , brlower , wlower  , mlower  , malower , mslower
     7            , mflower , dupper  , eupper  , seupper , bupper
     8            , shupper , vvupper , brupper , wupper  , mupper
     9            , maupper , msupper , mfupper , dmask   , dmaske
     1            , dmaskse , dmaskb  , dmasksh , dmaskvv , dmaskbr
     2            , dmaskw  , dmaskm  , dmaskma , dmaskms , dmaskmf
     3            , bmask   , bmaskd  , bmaske  , bmaskse , bmasksh
     4            , bmaskvv , bmaskbr , bmaskw  , bmaskm  , bmaskma
     5            , bmaskms , bmaskmf , nlrad   , radmin  , radmax
     6            , iradmm  , rorhrad , icnvlv  , bmajor  , bminor
     7            , bpa     , cpb     , eorb    , porf    , bworb
     8            , vlmin   , icut    , pcut    , pscale  , incpx
     9            , incpy   , units   , x1rdmn  , x1rdmx  , x2rdmn
     1            , x2rdmx  , x3rdmn  , x3rdmx  , iradbox
```

B.27 PGEN—Problem GENerator (subroutine aliased to PROBLEM)

This namelist is reserved for the problem generator, which sets the flow variables to the desired initial conditions. Thus the parameters which appear in this namelist depend on which problem is being studied. The desired problem is specified by setting the *EDITOR* alias `PROBLEM` in the file `zeus35.mac` to the name of the problem generating subroutine. This subroutine should initialise the active zones of all field variables as well as any inflow boundary arrays. The routines `bndyflgs` and `bndyall` are called after `PROBLEM`, and thus need not be called in the user's problem generator (§5.1).

Below is a description of the problem generator to `shkset`, which is used for the 1-D Brio and Wu problem and consistent with the sample of `dzeus35.s` given in §2.3. In general, the user will be writing their own problem generator and may, if they wish, call their namelist `pgen` as well. Note that it does not matter that more than one subroutine uses `pgen` as the name of its namelist, so long as only one problem generating subroutine is called (as is typical). If the user wishes to use one of the problem generators already in the `dzeus35` code, each of their namelists are described in the comments of the problem generating routine in exactly the same format as that for `shkset` which follows.

```
parameter                    description                          default

idirect    1 => 1-direction                          ie biggest => 1
           2 => 2-direction                          je biggest => 2
           3 => 3-direction                          ke biggest => 3
isetbdy    1 => set boundary conditions based on v(par)              1
           0 => inflow BC are already set by iib, etc.
n0         number of zones to be initialised.  The namelist is       nx1z
           repeatedly read from logical unit ioin until a total
           of ie-is+1 (or je-js+1, or ke-ks+1) zones are
           initialised.
d0         input density                                             tiny
e10        input internal energy density   ( = e1 )                 tiny
e1od0      input specific internal energy  ( = e1/d )                tiny
e20        input internal energy2 density  ( = e2 )                  tiny
e2od0      input specific internal energy2 ( = e2/d )                tiny
v10        input velocity in 1 direction                             0.0
v20        input velocity in 2 direction                             0.0
v30        input velocity in 3 direction                             0.0
b10        input magnetic field in 1 direction                       0.0
b20        input magnetic field in 2 direction                       0.0
b30        input magnetic field in 3 direction                       0.0

       namelist / pgen     /
     1               idirect , isetbdy , n0      , d0      , e10
     2             , e1od0   , e20     , e2od0   , v10     , v20
     3             , v30     , b10     , b20     , b30
```

C The *ZEUS-3D* variables

This Appendix contains a glossary of the variables used in `dzeus35`, and is meant to aid the user in writing subroutines and making changes to the source code itself. It is by no means complete, but should contain the variables needed for most purposes. All these variables are declared in the common deck `comvar`. Thus, adding the *EDITOR* command `*call comvar` before the local declarations makes all these variables accessible from within the subroutine.

The user should be aware of the index convention used. A 3-D array, such as the density, is denoted `d(i,j,k)`, where `i` is the index for the `x1` coordinate, `j` is the index for the `x2` coordinate, and `k` is the index for the `x3` coordinate. The coordinates `x1`, `x2`, and `x3` are intentionally generic, since an attempt has been made to write the code in a covariant fashion. In Cartesian, cylindrical, and spherical polar coordinates, `(x1,x2,x3)` corresponds to `(x,y,z)`, `(z,r,`ϕ`)` [not `(r,`ϕ`,z)`], and `(`ρ`,`θ`,`ϕ`)` respectively. In *FORTRAN*, the index which changes the fastest is the first one. Thus, in triple do-loops which manipulate the 3-D arrays, it is best to have the outer loop run on `k`, the middle loop run on `j`, and the inner loop run on `i`. If one of the directions is divided into more zones than the other two, then it is best that this direction be the 1-direction (with index `i`) since it is the inner loop which vectorises on vector machines. In Cartesian coordinates, this can always be arranged. The indices strictly follow a right-hand rule. Thus, the array `nijb(k,i)` is a 2-D array which has `k` as its first index and `i` as its second (and not `i` as the first index and `k` as the second which would follow a left-hand rule). In the tables in this appendix, arrays are given with their indexing to remind the user of the *ZEUS-3D* convention.

The user should also be aware of the gridding. The computational domain is divided into `in` by `jn` by `kn` zones. In each direction, five of these zones are "ghost" or "boundary" zones, while the remaining zones are "active" zones in which the equations of MHD are solved. In Cartesian geometry, these zones are rectangular boxes. In general, the gridding need not be uniform, so the ratio of the dimensions of each zone need not be constant across the grid. There are eight locations one can associate uniquely with each zone. Each of these locations can be tagged with the indices `(i,j,k)`. These locations are: the centre of each box, the centre of three of the six faces, the centre of three of the twelve edges, and one of the eight corners. In *ZEUS-3D*, there are two grids which are referred to as the half-grid (or the a-grid) and the full grid (or the b-grid). By convention, the `(i,j,k)`th point on the a-grid is half a grid spacing closer in each dimension to the origin than the `(i,j,k)`th point on the b-grid. Points on the b-grid `(x1b(i),x2b(j),x3b(k))` correspond to zone centres while points on the a-grid `(x1a(i),x2a(j),x3a(k))` correspond to zone corners.

Edges and faces have mixed grid coordinates. The centre of the 1-face has coordinates `(x1a(i),x2b(j),x3b(k))`, the centre of the 2-face has coordinates `(x1b(i),x2a(j),x3b(k))`, and the centre of the 3-face has coordinates `(x1b(i),x2b(j),x3a(k))`. The centre of the 1-edge has coordinates `(x1b(i),x2a(j),x3a(k))`, the centre of the 2-edge has coordinates `(x1a(i),x2b(j),x3a(k))`, and the centre of the 3-edge has coordinates `(x1a(i),x2a(j),x3b(k))`.

Part of the strength of *ZEUS-3D* is its use of a "staggered" grid. On such a grid, not all variables are located at the same place. Scalars (density and internal energy) are zone-centred quantities while the components of the flow vectors (velocity and magnetic field) are face-centred quantities penetrating the face upon which they are centred. Vectors derived

from vector quantities such as the current density ($\nabla \times \mathbf{B}$) and the induced electric field ($\mathbf{v} \times \mathbf{B}$) have edge-centred components parallel to the edges while scalars derived from vector quantities such as $\nabla \cdot \mathbf{v}$ are zone-centred. Thus, the two grids play equally important roles, and the user needs to be careful about which grid should be used and where the variables are located while making any changes to the code.

C.1 Grid variables

Limits for do-loops are tabulated below.

Variable	Description
`is, ie`	beginning and ending `i`-index for active zone-centres
`js, je`	beginning and ending `j`-index for active zone-centres
`ks, ke`	beginning and ending `k`-index for active zone-centres

Corresponding to each variable (`is`, `ie`, *etc.*) are the limiting variables (`ismn`, `iemx`, *etc.*) which indicate the extreme values possible for the do-loop indices should the grid extending option be used (§B.17). In addition, the variables `ism2`, `ism1`, `isp1`, `isp2`, and `isp3` exist which are set to `is-2`, `is-1`, `is+1`, `is+2`, and `is+3` respectively. If the computation is symmetric in the `i`-direction, `ism2`, `ism1`, `isp1`, `isp2`, and `isp3` are simply set to `is`. Similar variables exist for `ie`, `js`, `je`, `ks`, and `ke`.

In order to make the grid covariant, metric factors have been introduced which carry all the dependence of the geometry. In general, the metric appears in the expression for a differential in volume, $dV = g_1 dx_1\, g_2 dx_2\, g_3 dx_3$. In Cartesian coordinates, $g_1 = g_2 = g_3 = 1$. In cylindrical coordinates, $g_1 = g_2 = 1$, $g_3 = x_2$. In spherical polar coordinates, $g_1 = 1$, $g_2 = x_1$, $g_3 = x_1 \sin x_2$. Note that if one is limited to `XYZ`, `ZRP`, and `RTP` coordinates, there is no need for g_1 and g_3 can be split into two variables, one dependent just on x_1, the other just on x_2. In this way, g_3 can be represented by two 1-D arrays (g_{31} and g_{32}) rather than one 2-D array. Thus, three 1-D metric factors are used in *ZEUS-3D*.

The most commonly used a-grid variables are tabulated below.

Variable	Location	Description
`x1a(i)`	zone-corner	`x1`-coordinate in grid units
`x2a(j)`	zone-corner	`x2`-coordinate in grid units
`x3a(k)`	zone-corner	`x3`-coordinate in grid units
`dx1a(i)`	1-edge	`x1a(i+1) - x1a(i)`
`dx2a(j)`	2-edge	`x2a(j+1) - x2a(j)`
`dx3a(k)`	3-edge	`x3a(k+1) - x3a(k)`
`g2a(i)`	zone-corner	`= 1` for Cartesian and cylindrical coordinates, `= x1a(i)` for spherical polar coordinates
`g31a(i)`	zone-corner	`= g2a(i)`
`g32a(j)`	zone-corner	`= 1` for Cartesian coordinates, `= x2a(j)` for cylindrical coordinates, `= sin(x2a(j))` for spherical polar coordinates

The most commonly used b-grid variables are tabulated below.

Variable	Location	Description
`x1b(i)`	zone-centre	`x1`-coordinate in grid units
`x2b(j)`	zone-centre	`x2`-coordinate in grid units (radians in spherical polar coordinates)
`x3b(k)`	zone-centre	`x3`-coordinate in grid units (radians in both cylindrical and spherical polar coordinates)
`dx1b(i)`	1-face	`x1b(i) - x1b(i-1)`
`dx2b(j)`	2-face	`x2b(j) - x2b(j-1)`
`dx3b(k)`	3-face	`x3b(k) - x3b(k-1)`
`g2b(i)`	zone-centre	`= 1` for Cartesian and cylindrical coordinates, `= x1b(i)` for spherical polar coordinates
`g31b(i)`	zone-centre	`= g2b(i)`
`g32b(j)`	zone-centre	`= 1` for Cartesian coordinates, `= x2b(j)` for cylindrical coordinates, `= sin(x2b(j))` for spherical polar coordinates

Every grid variable has a corresponding inverse variable, denoted by appending an "`i`" to the variable name. Thus, `dx1ai=1/dx1a`, `x2bi=1/x2b`, *etc.* Evidently, there are numerous grid variables. However, only the a-grid variables `x1a`, `x2a`, and `x3a` are written to the restart dump. All others are re-computed when a job be resumed.

Note that `x1a(i) < x1b(i)`. The exact relationship between the two grids is:

```
x1b(i) = x1a(i) + 0.5 * dx1a(i)
```

with similar expressions applying for the 2- and 3-directions.

C.2 Field variables (3-D arrays)

The main field variables and their locations are as follows:

Variable	Location	Description
`d (i,j,k)`	zone centre	density
`v1(i,j,k)`	1-face	velocity in the 1-direction (grid units)
`v2(i,j,k)`	2-face	velocity in the 2-direction (grid units)
`v3(i,j,k)`	3-face	velocity in the 3-direction (grid units)
`e1(i,j,k)`	zone centre	first internal energy density ($\propto$ pressure)
`e2(i,j,k)`	zone centre	second internal energy density
`gp(i,j,k)`	zone-centre	gravitational potential
`b1(i,j,k)`	1-face	magnetic field in the 1-direction ($\mu_0 = 1$)
`b2(i,j,k)`	2-face	magnetic field in the 2-direction ($\mu_0 = 1$)
`b3(i,j,k)`	3-face	magnetic field in the 3-direction ($\mu_0 = 1$)

There is very little internal scaling of variables in *ZEUS-3D* that the user must consider. Density, energy, and velocity all may be scaled according to the user's needs simply by setting

the initial conditions as appropriate. For example, the user may wish to set the density and the sound speed at infinity to 1. This, along with some canonical length scale will set the time scale for the calculation. The only scaling implicit to *ZEUS-3D* is the permeability of free space ($4\pi \times 10^{-7}$ in mks, 4π in cgs) is set to 1. Thus, the total pressure (thermal plus magnetic) is given by $p_{\rm tot} = p_{\rm th} + B^2/2$. Having set the scale of the hydrodynamical variables, the user should set the magnetic fields with this additional scaling in mind.

If the *EDITOR* macro `ISO` is defined, the first internal energy, `e1`, is not declared. The second internal energy (`e2`), the gravitational potential (`gp`), and the magnetic field components (`b1`, `b2`, `b3`) are declared only if the *EDITOR* macros `TWOFLUID`, `GRAV`, and `MHD` are defined respectively. If `PSGRAV` is defined, an additional "pseudo-gravitational potential" array [`psgp(i,j,k)`] distinct from `gp` becomes available.

C.3 Boundary variables (2-D arrays)

Table for the first, second, and third inner-i boundaries follow:

Variable		Location	Description
`niib`	`(j,k)`		boundary type for all variables except `gp`
`giib`			boundary type for gravitational potential
`diib1`	`(j,k)`	zone-centre at `i=is-1`	density
`v1iib1`	`(j,k)`	1-face at `i=is`	1-velocity (normal to the boundary)
`v2iib1`	`(j,k)`	2-face at `i=is-1`	2-velocity (tangential to the boundary)
`v3iib1`	`(j,k)`	3-face at `i=is-1`	3-velocity (tangential to the boundary)
`e1iib1`	`(j,k)`	zone-centre at `i=is-1`	first internal energy density ($\propto$ pressure)
`e2iib1`	`(j,k)`	zone-centre at `i=is-1`	second internal energy density
`gpiib`	`(j,k)`	zone-centre at `i=is-1`	gravitational potential
`b2iib1`	`(j,k)`	2-face at `i=is-1`	2-magnetic field (tangential to the boundary)
`b3iib1`	`(j,k)`	3-face at `i=is-1`	3-magnetic field (tangential to the boundary)
`emf1iib1`	`(j,k)`	1-edge at `i=is-1`	1-*emf* (normal to the boundary)
`emf2iib1`	`(j,k)`	2-edge at `i=is`	2-*emf* (tangential to the boundary)
`emf3iib1`	`(j,k)`	3-edge at `i=is`	3-*emf* (tangential to the boundary)
`diib2`	`(j,k)`	zone-centre at `i=is-2`	density
`v1iib2`	`(j,k)`	1-face at `i=is-1`	1-velocity (normal to the boundary)
`v2iib2`	`(j,k)`	2-face at `i=is-2`	2-velocity (tangential to the boundary)
`v3iib2`	`(j,k)`	3-face at `i=is-2`	3-velocity (tangential to the boundary)
`e1iib2`	`(j,k)`	zone-centre at `i=is-2`	first internal energy density ($\propto$ pressure)
`e2iib2`	`(j,k)`	zone-centre at `i=is-2`	second internal energy density
`b2iib2`	`(j,k)`	2-face at `i=is-2`	2-magnetic field (tangential to the boundary)
`b3iib2`	`(j,k)`	3-face at `i=is-2`	3-magnetic field (tangential to the boundary)
`emf1iib2`	`(j,k)`	1-edge at `i=is-2`	1-*emf* (normal to the boundary)
`v1iib3`	`(j,k)`	1-face at `i=is-2`	1-velocity (normal to the boundary)

Note there is no second gravitational potential boundary array. Analogous boundary variables exist at the outer-`i` boundary (`oib`), inner-`j` boundary (`ijb`), outer-`j` boundary (`ojb`),

inner-k boundary (ikb), and outer-k boundary (okb). Note that the i-boundary variables use indices (j,k) and are declared so long as the *EDITOR* macro ISYM is *not* defined. Similarly, the j-boundary variables use indices (k,i) and are declared so long as JSYM is *not* defined while the k-boundary variables use indices (i,j) and are declared so long as KSYM is *not* defined. All internal energy boundary variables (e1iib1, *etc.*) are *not* declared if ISO is defined. The boundary variables for the second internal energy (e2iib1, *etc.*), gravity (gpiib, *etc.*), and the magnetic field components (b2iib1, *etc.*) are declared *only if* TWOFLUID, GRAV, and MHD are defined respectively. Note that boundary variables are used only for regions of the boundary specified as inflow [niib(j,k)=8 or 10, and/or giib=3]. For boundary type 8 (selective inflow), grid values are used where boundary variables are set to huge. For the gravitational potential, the boundary variable, gpiib=3, is set to known analytical or asymptotic values. For all other boundary types, the boundary values of the flow variables are determined from the values in the neighbouring active zones (§B.8).

C.4 Scratch variables

There are a multitude of scratch arrays available which can be used to minimise the additional memory required by the user's subroutines. These should be used wherever possible, especially for 3-D arrays. There are 26 1-D arrays dimensioned (ijkx) and named wa1d through wz1d. There are 15 2-D arrays dimensioned (idim,jdim) and named wa2d through wo2d [plus an additional six "transpose" arrays dimensioned (jdim,idim) and named wa2dt through wf2dt]. See §C.6 for the definition of the parameters idim and jdim. Finally, there are eight 3-D arrays dimensioned (in,jn,kn) and named wa3d through wh3d.

C.5 Sundry variables (an abbreviated list)

Variable	Description
ioin	logical unit attached to input deck
iolog	logical unit attached to message log file
iotty	logical unit attached to terminal (TTY or CRT)
iodmp	logical unit attached to restart dumps
ioplt1	logical unit attached to 1-D plot files
ioplt2	logical unit attached to 2-D plot files
iopix	logical unit attached to 2-D pixel dumps
iovox	logical unit attached to 3-D voxel dumps
iousr	logical unit attached to user dumps
iotsl	logical unit attached to time slice ascii dump
iotslp	logical unit attached to time slice plot dump
iodis	logical unit attached to display dump
iorad	logical unit attached to *RADIO* dump
nhy	number of cycles (time steps) completed in simulation
nwarn	running total of warnings issued
prtime	problem time elapsed in simulation
dt	increment of problem time that solution is being advanced

In addition, all of the namelist variables (except for namelist `pgen`) are declared in `comvar`.

C.6 Parameters

Primary parameters (those which the user can set) include:

Parameter	Description
`in`	number of zones in 1-direction plus 5 ghost zones
`jn`	number of zones in 2-direction plus 5 ghost zones
`kn`	number of zones in 3-direction plus 5 ghost zones
`nxpx`	maximum number of pixels in the x-direction for pixel dumps
`nypx`	maximum number of pixels in the y-direction for pixel dumps
`nxrd`	maximum number of pixels in the x-direction for *RADIO* dumps
`nyrd`	maximum number of pixels in the y-direction for *RADIO* dumps
`niov`	maximum number of variables plotted/dumped
`nios`	maximum number of slices for each variable plotted/dumped
`ncls`	maximum number of contour levels in 2-D *NCAR/PSPLOT* plots
`ntsl`	maximum number of time slices to be collected for plots
`pi`	3.14159...
`nmat`	maximum number of materials. With `TWOFLUID` set, this should be 2
`isig`	number of significant figures to which some real*8 numbers are rounded.
`tiny`	1.0×10^{-99}: smallest greater-than-zero number available on machine
`huge`	$1.0 \times 10^{+99}$: largest number available on machine
`smll`	1.0×10^{-6}: a convenient "small" number.
`lrge`	$1.0 \times 10^{+6}$: a convenient "large" number.

The parameter `nios` is used by the following I/O formats: 1-D *NCAR/PSPLOT* plots, 2-D *NCAR/PSPLOT* plots, pixel dumps, and display dumps. The parameter `niov` is used by all these I/O formats, plus: voxel dumps, *HDF* dumps, and *RADIO* dumps. They are both currently set to 20 in the common deck `par`, and can be altered as needed.

Secondary parameters (those which are computed from the primary parameters and the user does not set but should still be aware of):

Parameter	Description
`ijkx`	the maximum of `in`, `jn`, and `kn`
`ijkn`	the minimum of `in`, `jn`, and `kn`
`idim`	= `jn` (`kn`, `in`) if `ISYM` (`JSYM`, `KSYM`) is set [1- (2-, 3-) symmetry flag] = `ijkx` if no symmetry is set
`jdim`	= `kn` (`in`, `jn`) if `ISYM` (`JSYM`, `KSYM`) is set [1- (2-, 3-) symmetry flag] = `ijkx` if no symmetry is set