Use of FWEB in DEGAS 2 to Expedite Documentation and Maintainability

D. P. Stotler and C. F. F. Karney

CPPG Seminar

June 13, 2000

Introduction

- John Krommes' FWEB package (based on Knuth's WEB) combines code & documentation in a single file.
  - Encourages code author to document code as it's being written.
- But, FWEB's preprocessor permits creation of powerful macros,
  - Can substantially improve programming environment over plain FORTRAN,
  - Simplifying code writing and maintenance.
- This talk *not* intended as an FWEB tutorial,
- Rather, will use DEGAS 2 to show what FWEB can do.
  - Credit to Charles for most of these examples.

Outline

1. Simple uses of FWEB
   (a) An example code,
   (b) Equations in documentation,
   (c) Use of modules for organizing code,
   (d) Short macros.
2. More complicated applications of FWEB preprocessor
   (a) Creation of “class” files combining variable declarations and documentation,
      - “Object”-like entities in DEGAS 2.
   (b) Use FWEB macros to facilitate code maintenance,
      i. Common blocks,
      ii. Dynamic memory tasks,
      iii. Reading, writing netCDF files,
      iv. MPI broadcast of common block data.

It's easy to get started with FWEB.
Here's a simple example.

```
% $Id: $   % Picks up identifying string from CVS
\Title{A test file}   % Specify a title for this document.
% $Log: $  % Picks up log information from CVS

@* Introduction. In this (named) section you can provide introductory
text (in \TeX\ form) for the code. This example file contains just a few
things beyond those described in John's \FWEB\ example. The macros
defined below allow you to eliminate unnamed constants from the code.
Header files with other macros can also be included.

\ID{$Id: $}
@m FILE 'template.web'
@m NMAX 35
@I macros.hweb

@ This is an unnamed section and does not generate an entry in the table
of contents. The \verb+@@a+ command denotes the start of the code. In
a typical DEGAS 2 \FWEB\ file, all of the code is accumulated under
a module called ''Functions and subroutines''. You can specify additional
information here about the program, etc.

@a

      program main
      call compute         // This is a short C-style comment
      stop
      end

      @<Functions and subroutines@>

@ A subroutine. Add subroutines like this.

@<Functions...@>=

      subroutine compute
/*
   Longer C-style comments can also be handled.
*/
      write (*,*) ' FWEB will take care of wrapping FORTRAN lines at char-
acter 72'
      write (*,*) ' nmax = ',NMAX
      return
      end

@* INDEX.
```

When you run FTANGLE on the file, you get compilable code:

```
C* 30: *
*line 182 "/u/dstotler/degas2/src/macros.hweb"
```

```
*line 21 "/u/dstotler/degas2/src/test.web"
C* :30 *
C* 31: *
*line 29 "/u/dstotler/degas2/src/test.web"

      program main
      call compute
      stop
      end

C* 32: *
*line 39 "/u/dstotler/degas2/src/test.web"

      subroutine compute
      write(*,*)' FWEB will take care of wrapping FORTRAN lines at chara
     &cter 72'
      write(*,*)' nmax = ',35
      return
      end

C* :32 *
*line 34 "/u/dstotler/degas2/src/test.web"

C* :31 *

c     $Id: $
c     $Log: $
```

Running FWEAVE gives a TEX file, which can be typeset:

1 Introduction

In this (named) section you can provide introductory text (in TEX form) for the code. This example file contains just a few things beyond those described in John's FWEB example. The macros defined below allow you to eliminate unnamed constants from the code. Header files with other macros can also be included.

```
$Id:  $
```

"test.f" 1 ≡

```
@m FILE 'template.web'
@m NMAX 35
```

This is an unnamed section and does not generate an entry in the table of contents. The @a command denotes the start of the code. In a typical DEGAS 2 FWEB file, all of the code is accumulated under a module called "Functions and subroutines". You can specify additional information here about the program, etc.

"test.f" 1.1 ≡

```
program main
  call compute    // This is a short C-style comment
  stop
end
```

⟨ Functions and subroutines 1.2 ⟩

A subroutine. Add subroutines like this.

⟨ Functions and subroutines 1.2 ⟩ ≡

```
subroutine compute    /* Longer C-style comments can also be handled. */
  write(*, *) ' FWEB will take care of wrapping FORTRAN lines at character 72'
  write(*, *) ' nmax = ', NMAX
  return
end
```

This code is used in section 1.1.

By integrating documentation into one file, see correspondence between equations and code:

This routine is a single argument function which provides the integrand in a calculation velocity-weighted reaction rates for cases involving two "heavy" particles. The plasma species is assumed to be described by a Maxwellian distribution function. The desired integral can be expressed in terms of[1]

$$I_{\ell,n} \equiv \frac{1}{\sqrt{\pi} u u_p^{n+1}} \int_0^\infty dv_{rel}\, v_{rel}^{2+n} \sigma_\ell(v_{rel}) \left\{ \exp\left[\frac{-(v_{rel}-u)^2}{u_p^2}\right] - (-1)^n \exp\left[\frac{-(v_{rel}+u)^2}{u_p^2}\right] \right\}, \qquad (5)$$

where u is the velocity of the neutral particle, and u_p is the thermal velocity of the plasma particle. The latter is related to the plasma temperature by $T_p = \frac{1}{2} m_p u_p^2$. The result provided by this routine is $I_{\ell,n} u_p^n$ and, thus, has the same dimensions as $\langle \sigma v v^n \rangle$.

This routine is set up with a single argument so that it can be used with "canned" integration routines; the argument *rel_velocity* is the relative velocity v_{rel} in the above expressions. The factors u and u_p, as well as the exponent n is represented by the variable *moment_number*. Also in common is the mass *ra_mass* used to convert the relative velocity into an energy for use in evaluating the cross section *sigma*. The value to be used for *ra_mass* may vary, depending on the origin of the data used in *sigma*.

The units of the various velocities are taken to be $\sqrt{\text{eV/amu}}$ so that values of order unity for *rel_velocity* are reasonable. It is assumed that *ra_mass* has units of grams, that the argument of *sigma* should be in eV, and that *sigma* itself is in cm^2. The factor *multiplier* is defined so that the resulting integral has cgs units.

```
⟨ Functions and subroutines 2 ⟩ +≡
    function ion_rate_integrand(rel_velocity)
      implicit_none_f77
      ra_common    // Common
      implicit_none_f90
      real ion_rate_integrand
      real rel_velocity    // Input

      real zbracket, multiplier, ze, exponent_minus,    /* Local */
        exponent_plus, zsigma
      real sigma    // External function

      ra_localcommon    // Local common

      ze = half * (ra_mass / atomic_mass_unit_g) * rel_velocity**2
      zsigma = sigma(ze)
      exponent_plus = -(rel_velocity + u)**2 / up**2
      exponent_minus = -(rel_velocity - u)**2 / up**2
      zbracket = exp(exponent_minus) - (-1)**moment_number * exp(exponent_plus)
      multiplier = one / (sqrt(PI) * u * up) * (sqrt(ev_to_ergs / atomic_mass_unit_g))**1+moment_number

      ion_rate_integrand = multiplier * rel_velocity**2+moment_number * zsigma * zbracket

      return
    end
```

You can use FWEB modules to break up code without using subroutines:

```
if (keyword ≡ 'dg_file') then
    ⟨ Process DG File 2 ⟩
  else if (keyword ≡ 'sonnet_mesh' ∨ keyword ≡ 'uedge_mesh') then
    assert(next_token(line, b, e, p))
    open_file(diskin2, line(b : e))    /* E.g., sonnet_mesh filename 120 24 */
    if (keyword ≡ 'sonnet_mesh') then
      assert(next_token(line, b, e, p))
      nxd = read_integer(line(b : e))
      assert(next_token(line, b, e, p))
      nzd = read_integer(line(b : e))
    else if (keyword ≡ 'uedge_mesh') then
      assert(next_token(line, b, e, p))
      read (diskin2, *)    // First line is a comment
      read (diskin2, *) nxd
      read (diskin2, *) nzd
    end if
    var_alloc(mesh_xz)
    if (keyword ≡ 'sonnet_mesh') then
      call read_sonnet_mesh(diskin2, nxd, nzd, mesh_xz)
    else if (keyword ≡ 'uedge_mesh') then
      call read_uedge_mesh(diskin2, nxd, nzd, mesh_xz)
    end if
  else if (keyword ≡ 'wallfile') then
    ⟨ Read Wall File 3 ⟩
  else if (keyword ≡ 'print_min_max') then
    call find_min_max(num_nodes, nodes, node_type, x_min, x_max, z_min, z_max)
    write (stdout, *) ' X range = ', x_min, ' -> ', x_max
    write (stdout, *) ' Z range = ', z_min, ' -> ', z_max
  else if (keyword ≡ 'bounds') then
    ⟨ Set Bounds 1.4 ⟩
  else if (keyword ≡ 'symmetry') then
    ⟨ Set Symmetry 1.5 ⟩
  else if (keyword ≡ 'print_walls') then
    assert(next_token(line, b, e, p))
    file_format = line(b : e)
    if (next_token(line, b, e, p)) then
      walloutfile = line(b : e)
      nunit = diskout
    else
      walloutfile = char_undef
      nunit = stdout
    end if
    call print_walls(nunit, file_format, walloutfile, num_walls, wall_element_count, wall_nodes, nodes)
  else if (keyword ≡ 'end_prep') then
    ⟨ End Prep 1.6 ⟩
  else if (keyword ≡ 'new_zone') then
    zone++
```

2 Read and process DG file

The code for each of the modules then appears later in the same file:

⟨ Process DG File 2 ⟩ ≡

```
assert(next_token(line, b, e, p))
open_file(diskin2, line(b : e))    /* Read DG's .dgo file */
ielement = 0
section = sec_undef    /* Assume that if the file has a ".dgo" that it came from DG and needs to be
    converted from mm to meters. Assume that it is otherwise in meters. */
if (index(line(b : e), '.dgo') > 0) then
  mult = const(1., −3)
else
  mult = one
end if
dg_loop: continue
if (read_string(diskin2, line, length)) then
  assert(length ≤ len(line))
    /* We do not want to have to manually change the format of the files DG writes. The node
    coordinates are currently comma-delimited (with possible additional spaces). Replace the
    commas in the current line with spaces so we can use our usual string utilities. */
  do i = 1, length
    if (line(i : i) ≡ ',')
      line(i : i) = '␣'
  end do
  length = parse_string(line( : length))
  p = 0
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ 'p1') then
    section = sec_p1
  else if (line(b : e) ≡ 'p2') then
    section = sec_p2
  else if (line(b : e) ≡ 'misselem') then
    section = sec_miss
      /* ADD SECTION TO READ jedgi1, jedgo2, jedgo1, jedgi2 SECTIONS AND PROCESS */
  else if (line(b : e) ≡ 'finish') then
    section = sec_done
  else
    if (section ≡ sec_p1 ∨ section ≡ sec_p2) then
      temp_x = read_real_soft_fail(line(b : e))
      if (temp_x ≠ real_undef) then    // Failure
        assert(next_token(line, b, e, p))
        temp_z = read_real_soft_fail(line(b : e))
      end if
      if (temp_x ≡ real_undef ∨ temp_z ≡ real_undef) then
        section = sec_undef
        go to dg_loop    // Can trash this line since it did not match anything we've planned for
      end if
      temp_x *= mult
      temp_z *= mult
      inode = 0
```

FWEB's macros permit elimination of hardwired constants. Note usage of "const" macro for flexible precision.

2 Header macros for constants

[/u/dstotler/degas2/src/constants.hweb]

[/u/dstotler/degas2/src/constants.hweb] Macro definitions.

```
$Id:  constants.hweb,v 1.1 1993/09/23 12:17:11 karney Exp $
```

Definitions of some physical constants, taken from "Physics Vade Mecum".

"test.f" 2.1 ≡

```
@m speed_of_light const(2.99792458, 8)     // c(m/s)
@m electron_charge const(1.60217733, −19)     // e(C)
@m plancks_const const(6.6260755, −34)     // h(J s)
@m electron_mass const(9.1093897, −31)     // m_e(kg)
@m proton_mass const(1.6726231, −27)     // m_p(kg)
@m neutron_mass const(1.6749286, −27)     // m_n(kg)
@m atomic_mass_unit const(1.6605402, −27)     // m_u(kg)
@m boltzmanns_const const(1.380658, −23)     // k(J/K)
```

Macros can be used for small, repeated segments of code. In DEGAS 2, some of these represent "methods" associated with the "classes".

- **DEGAS 2 has several macros devoted to handling "ragged arrays".**
- **Note usage of "assert" macro for catching unexpected code behavior (essential when working with pointers).**

[/u/dstotler/degas2/src/background.hweb] Move velocity from external reference plane to internal position.

"classes.f" 20.2 ≡

```
@m v_ext_to_int(x, v, v_t, sym, coord)
      if (sym ≡ geometry_symmetry_plane ∨ sym ≡ geometry_symmetry_oned ∨ (sym ≡
          geometry_symmetry_none ∧ coord ≡ plasma_coords_cartesian) ∨ (x1^2 + x2^2 ≡ zero)) then
        vc_copy(v, v_t)
      else if (sym ≡ geometry_symmetry_cylindrical ∨ (sym ≡ geometry_symmetry_none ∧ coord ≡
          plasma_coords_cylindrical)) then
        assert(x1^2 + x2^2 > zero)
        v_t1 = (v1 * x1 − v2 * x2) / sqrt(x1^2 + x2^2)
        v_t2 = (v1 * x2 + v2 * x1) / sqrt(x1^2 + x2^2)
        v_t3 = v3
      else
        assert(F)
      end if
```

[/u/dstotler/degas2/src/background.hweb] Move velocity from internal position external reference plane.

"classes.f" 20.3 ≡

```
@m v_int_to_ext(x, v, v_t, sym, coord)
      if (sym ≡ geometry_symmetry_plane ∨ sym ≡ geometry_symmetry_oned ∨ (sym ≡
          geometry_symmetry_none ∧ coord ≡ plasma_coords_cartesian)) then
        vc_copy(v, v_t)
      else if (sym ≡ geometry_symmetry_cylindrical ∨ (sym ≡ geometry_symmetry_none ∧ coord ≡
          plasma_coords_cylindrical)) then
        assert(x1^2 + x2^2 > zero)
        v_t1 = (v1 * x1 + v2 * x2) / sqrt(x1^2 + x2^2)
        v_t2 = (−v1 * x2 + v2 * x1) / sqrt(x1^2 + x2^2)
        v_t3 = v3
      else
        assert(F)
      end if
```

[/u/dstotler/degas2/src/sysdep.hweb] Define precision.

Can use macros to make compile-time decisions. E.g., for cross-platform compatibility, F90 F77, . . .

"test.f" 3.3 ≡
  @f *single_precision real*
  @f *address real*

@#if *FORTRAN90*
  @m *f90_kinds SINGLE* = *kind*(0.0), *DOUBLE* = *selected_real_kind*(12)
  @m *real* **REAL**(*kind* = *DOUBLE*)
  @m areal(*x*) **REAL**(*x*, *DOUBLE*)
  @m *const*(*x*, . . .) *x***@&$IFCASE (#0,,e###1)@&**_*DOUBLE*
  @m *epsilon* (5 ∗ *EPSILON*(*const*(0.0))) // 1.1e-15 on Sun/Alpha. 7.1e-14 on Cray.
  @m *single_precision* **REAL**(*kind* = *SINGLE*)
  @m *single*(*x*) **REAL**(*x*, *SINGLE*)
@#else // Use single precision on the Cray
@#if *CRAY*
  @m *real* **REAL**
  @m areal **REAL**
  @m *const*(*x*, . . .) *x***@&$IFCASE (#0,e0,e###1)**
  @m *epsilon* $1.0 \cdot 10^{-15}$
  @m *single_precision* Real
  @m *single* Real
@#else
  @m *real* *DOUBLEPRECISION* // Default precision
  @m areal *DBLE* // Coerce to default precision
  @m *const*(*x*, . . .) *x***@&$IFCASE (#0,d0,d###1)** // Constant in default precision
  @m *epsilon* $1.0 \cdot 10^{-15}$D // Approximate round-off error
  @m *single_precision* Real // Declare single precision
  @m *single* Real // Coerce to single precision
@#endif
@#endif

@#if *CRAY*
  @m *HIPREC* 1
@#else
  @m *HIPREC* 0
@#endif // Pointers are 64-bits on Alphas
@#if *ALPHA*
  @m *address* *INTEGER* ∗ 8
@#elif *SGI*
  @m *address* *INTEGER* ∗ 8
@#else
  @m *address* *INTEGER*
@#endif

@#if defined (*USEINT*)
  @m *USEINT* 1
@#else
@#if *SGI* ∨ *CRAY* ∨ *LINUX*
  @m *USEINT* 0
@#else

6 Species definitions

[/u/dstotler/degas2/src/species.hweb]

```
$Id:  species.hweb,v 1.11 1999/03/25 20:47:37 dstotler Exp $
```

More complicated examples . . .

All DEGAS 2 common variables (and some local) are defined in header files; facilitated by sophisticated macros. Permits:

- **Dimensioning information appears in one and only one place,**
- **Documentation on variables included in same file,**
- **Inclusion of macros for modification of appearance of variables in code,**
- **And definition of "methods" involving these variables.**

[/u/dstotler/degas2/src/species.hweb] Information about species is held in arrays in common blocks. An species is identified by an integer indexing into these arrays.

"classes.f" 6.1 ≡
@f *sp_decl integer*

@m *sp_args*(x) x

@m *sp_dummy*(x) x

@m *sp_decl*(x) **integer** x

@m *sp_copy*(x, y) $y = x$

@m *sp_check*(x) $(x > 0 \wedge x \leq sp\_num)$

@m *sp_name*(x) $species\_name_x$
@m *sp_sy*(x) $species\_sy_x$
@m *sp_m*(x) $species\_m_x$
@m *sp_z*(x) $species\_z_x$
@m *sp_ncomp*(x) $species\_ncomp_x$
@m *sp_generic*(x) $species\_generic_x$
@m *sp_multiplicity*(x) $species\_multiplicity_x$
@m *sp_el*(x, i) $species\_el_{i,\ x}$
@m *sp_count*(x, i) $species\_count_{i,\ x}$
@m *sp_lookup*(sy) $string\_lookup(sy,\ species\_sy,\ sp\_num)$

[/u/dstotler/degas2/src/species.hweb] Length specifications.

"classes.f" 6.2 ≡
@m *sp_sy_len* 8
@m *sp_name_len* 32
@m *sp_ncomp_max* 10

[/u/dstotler/degas2/src/species.hweb] Variable definitions.

"classes.f" 6.3 ≡
package_init(*sp*)
define_dimen_pk(*sp*, *species_symbol_string*, *sp_sy_len*)
define_dimen_pk(*sp*, *species_name_string*, *sp_name_len*)
define_var_pk(*sp*, *sp_num*, *INT*)
define_dimen_pk(*sp*, *species_ind*, *sp_num*)
define_dimen_pk(*sp*, *species_comp_ind*, *sp_ncomp_max*)
define_varp_pk(*sp*, *species_name*, *CHAR*, *species_name_string*, *species_ind*)
define_varp_pk(*sp*, *species_sy*, *CHAR*, *species_symbol_string*, *species_ind*)
define_varp_pk(*sp*, *species_m*, *FLOAT*, *species_ind*)
define_varp_pk(*sp*, *species_z*, *INT*, *species_ind*)
define_varp_pk(*sp*, *species_ncomp*, *INT*, *species_ind*)
define_varp_pk(*sp*, *species_generic*, *INT*, *species_ind*)
define_varp_pk(*sp*, *species_multiplicity*, *INT*, *species_ind*)
define_varp_pk(*sp*, *species_el*, *INT*, *species_comp_ind*, *species_ind*)
define_varp_pk(*sp*, *species_count*, *INT*, *species_comp_ind*, *species_ind*)
define_varlocal_pk(*sp*, *species_version*, *CHAR*, *string*)
package_end(*sp*)

7 Species class attribute descriptions

[/u/dstotler/degas2/src/species.hweb]

[/u/dstotler/degas2/src/species.hweb] Define species. Prefix is *sp*.

The identifier *species* is an integer variable.

sp_name(*species*) Returns the long name (length *sp_name_len*).

sp_sy(*species*) Returns the symbolic name (length *sp_sy_len*).

sp_m(*species*) Returns the mass (in kg).

sp_z(*species*) Returns the charge number.

sp_ncomp(*species*) Returns the number of components in a species (maximum size *sp_ncomp_max*).

sp_generic(*species*) Is the integer species number of the isotopic equivalent to *species* which has been designated in the species input as the archetype or "generic" member of that family.

sp_multiplicity(*species*) Number of different ways in which the isotopically equivalent elements in *species* could be arranged.

sp_el(*species*, *i*) Returns the element for the *i*th component.

sp_count(*species*, *i*) Returns the count of the *i*th component.

For some DEGAS 2 "classes", macros allow a group of variables to be handled as a single entity. The addition of macro "methods" gives these classes an almost object-like character. Here, "flight" (fl) is the stack of "particles" (pt) being followed. In turn, "particles" have attributes like "species" (sp), velocity, "location" (lc), weight, and time.

```
do i = 1, pr_rc_num(pt_test(fl_current(x)))
        sum = sum + rate_i
        if (sum ≥ rnd) then
          assert(rate_i > zero)
          call score_test(tl_est_collision, one / other_rate, pt_args(fl_current(x)), estimator_factors)
          pt_copy(fl_current(x), prod_0)
          call score_reaction(tl_est_collision,    /* And process! */
           one/rate_i, pr_reaction_args(i), rate_i, nprod, pt_args(prod_0), estimator_factors,
               rn_args(fl_rand(x)))
          goto break1
        end if
      end do
      assert(F)
break1: continue
      fl_pointer(x)--
      if (nprod > 0) then
        do i = 1, nprod
          if (pt_test(prod_i) > 0 ∧ pt_w(prod_i) > zero) then
            fl_pointer(x)++
            pt_copy(prod_i, fl_current(x))
          end if
        end do
      end if
      if (fl_pointer(x) ≤ 0)
        goto break
      assert(fl_check(x))
      goto loop
```

The resulting F77 file shows these "objects" broken down into sets of ordinary FORTRAN variables. Note that the macro arguments "x" and "prod" are used in naming the local variables!

```
      do i=1,problem_reaction_num(test_stack_x(pointer_x))
      sum=sum+rate(i)
      if(sum.GE.rnd)then
      if(rate(i).GT.0.0d0)continue

      call score_test(2,1.0d0/other_rate,species_stack_x(pointer_x),test
     &_stack_x(pointer_x),time_stack_x(pointer_x),weight_stack_x(pointer
     &_x),pos_stack_x(1,pointer_x),cell_stack_x(pointer_x),zone_stack_x(
     &pointer_x),surface_stack_x(pointer_x),cell_next_stack_x(pointer_x)
     &,zone_next_stack_x(pointer_x),sector_stack_x(pointer_x),sector_nex
     &t_stack_x(pointer_x),velocity_stack_x(1,pointer_x),type_stack_x(po
     &inter_x),author_stack_x(pointer_x),estimator_factors)

      species_prod(0)=species_stack_x(pointer_x)
      test_prod(0)=test_stack_x(pointer_x)
      time_prod(0)=time_stack_x(pointer_x)
      weight_prod(0)=weight_stack_x(pointer_x)
      velocity_prod(1,0)=velocity_stack_x(1,pointer_x)
      velocity_prod(2,0)=velocity_stack_x(2,pointer_x)
      velocity_prod(3,0)=velocity_stack_x(3,pointer_x)
      pos_prod(1,0)=pos_stack_x(1,pointer_x)
      pos_prod(2,0)=pos_stack_x(2,pointer_x)
      pos_prod(3,0)=pos_stack_x(3,pointer_x)
      cell_prod(0)=cell_stack_x(pointer_x)
      zone_prod(0)=zone_stack_x(pointer_x)
      surface_prod(0)=surface_stack_x(pointer_x)
      cell_next_prod(0)=cell_next_stack_x(pointer_x)
      zone_next_prod(0)=zone_next_stack_x(pointer_x)
      sector_prod(0)=sector_stack_x(pointer_x)
      sector_next_prod(0)=sector_next_stack_x(pointer_x)
      type_prod(0)=type_stack_x(pointer_x)
      author_prod(0)=author_stack_x(pointer_x)

      call score_reaction(2,1.0d0/rate(i),i,rate(i),nprod,species_prod(0

                    ⋮

      if(pointer_x.LE.0)goto 90007

      if((number_x.GE.0.AND.(source_x.GT.0.AND.source_x.LE.so_grps).AND.
     &pointer_x.GT.0.AND.pointer_x.LE.40))continue

      goto 90000
```

This routine shows use of FWEB macros for common blocks, reading netCDF files, and MPI broadcasting of common data.

Read in background data from netcdf file `background.nc`

```
⟨ Functions and Subroutines 1.2 ⟩ +≡
  subroutine nc_read_background
    implicit_none_f77
    zn_common
    bk_common
    so_common
    rf_common
    mp_common
    implicit_none_f90
    integer fileid
    character*FILELEN tempfile    // local

    bk_ncdecl
    nc_decls
    cm_ncdecl
    mp_decls
    ⟨ Memory allocation interface 0 ⟩
    so_ncdecl
    vc_decls

    if (mpi_master) then
      tempfile = filenames_array backgroundfile
      assert(tempfile ≠ char_undef)
      fileid = ncopen(tempfile, NC_NOWRITE, nc_stat)

      cm_ncread(fileid)
      bk_ncread(fileid)
      so_ncread(fileid)

      call ncclose(fileid, nc_stat)
    end if

    cm_mpibcast
    bk_mpibcast
    so_mpibcast

    return
  end
```

When expanded by FTANGLE, the common block macros look like this:

```
subroutine nc_read_background
implicit none

integer zn_num
common/zn_com_i/zn_num
integer zone_type_num((1):(4))

     ⋮            (34 lines total)

save/zn_com_c/
save/zn_com_i/
save/zn_com_p/

integer bk_num
common/bk_com_i/bk_num
integer background_coords
common/bk_com_i/background_coords
pointer(ptr_background_n,background_n)
DOUBLE PRECISION background_n((1):(bk_num),(1):*)
common/bk_com_p/ptr_background_n
pointer(ptr_background_v,background_v)
DOUBLE PRECISION background_v((1):(3),(1):(bk_num),(1):*)
common/bk_com_p/ptr_background_v
pointer(ptr_background_temp,background_temp)
DOUBLE PRECISION background_temp((1):(bk_num),(1):*)
common/bk_com_p/ptr_background_temp
save/bk_com_i/
save/bk_com_p/

integer so_grps
common/so_com_i/so_grps
integer so_seg_tot

     ⋮            (73 lines total)

save/so_com_c/
save/so_com_i/
save/so_com_r/
save/so_com_p/

pointer(ptr_filenames_array,filenames_array)
character*((96))filenames_array((1):*)
common/rf_com_p/ptr_filenames_array
save/rf_com_p/
```

And here are the expanded ncread macros. Note the memory allocation calls.

```
      integer fileid
      character*96 tempfile
      integer bk_num_id
      integer background_ind_id
          ⋮
      integer source_segment_rel_wt_id
      integer source_segment_prob_alias_id
      integer source_segment_ptr_alias_id

      DOUBLE PRECISION vector_temp(3)
      if(.TRUE.)then
      tempfile=filenames_array(2)
      if(tempfile.NE.'undefined')continue
      fileid=ncopn(tempfile,0,nc_stat)
      vector_id=ncdid(fileid,'vector',nc_stat)
      call ncdinq(fileid,vector_id,nc_dummy,nc_size,nc_stat)
      if(nc_size.EQ.((3)-(1)+1))continue
      string_id=ncdid(fileid,'string',nc_stat)
      call ncdinq(fileid,string_id,nc_dummy,nc_size,nc_stat)
      if(nc_size.EQ.((136)-(1)+1))continue

      bk_num_id=ncvid(fileid,'bk_num',nc_stat)
      call ncvinq(fileid,bk_num_id,nc_dummy,nc_type,nc_rank,nc_dims,nc_a
     &ttr,nc_stat)

      call ncvgt(fileid,bk_num_id,nc_corner,nc_edge,bk_num,nc_stat)
      background_ind_id=ncdid(fileid,'background_ind',nc_stat)
      call ncdinq(fileid,background_ind_id,nc_dummy,nc_size,nc_stat)
      if(nc_size.EQ.((bk_num)-(1)+1))continue
      bk_plasma_ind_id=ncdid(fileid,'bk_plasma_ind',nc_stat)
      call ncdinq(fileid,bk_plasma_ind_id,nc_dummy,nc_size,nc_stat)
      if(nc_size.EQ.((zone_type_num(2))-(1)+1))continue
      background_coords_id=ncvid(fileid,'background_coords',nc_stat)
      call ncvinq(fileid,background_coords_id,nc_dummy,nc_type,nc_rank,n
     &c_dims,nc_attr,nc_stat)
      call ncvgt(fileid,background_coords_id,nc_corner,nc_edge,backgroun
     &d_coords,nc_stat)
      background_n_id=ncvid(fileid,'background_n',nc_stat)
      call ncvinq(fileid,background_n_id,nc_dummy,nc_type,nc_rank,nc_dim
     &s,nc_attr,nc_stat)
      if(nc_dims(1).EQ.background_ind_id)continue
      if(nc_dims(2).EQ.bk_plasma_ind_id)continue
      ptr_background_n=mem_alloc((((bk_num)-(1)+1)*((zone_type_num(2))-
(
```

```
&1)+1)))
 nc_corner(1)=1
 nc_edge(1)=((bk_num)-(1)+1)
 nc_corner(2)=1
 nc_edge(2)=((zone_type_num(2))-(1)+1)
 call ncvgt(fileid,background_n_id,nc_corner,nc_edge,background_n,n
&c_stat)
 background_v_id=ncvid(fileid,'background_v',nc_stat)
 call ncvinq(fileid,background_v_id,nc_dummy,nc_type,nc_rank,nc_dim
&s,nc_attr,nc_stat)
 if(nc_dims(1).EQ.vector_id)continue
 if(nc_dims(2).EQ.background_ind_id)continue
 if(nc_dims(3).EQ.bk_plasma_ind_id)continue

      ⋮            (62 lines total)

 call ncvgt(fileid,background_temp_id,nc_corner,nc_edge,background_
&temp,nc_stat)


 so_grps_id=ncvid(fileid,'so_grps',nc_stat)
 call ncvinq(fileid,so_grps_id,nc_dummy,nc_type,nc_rank,nc_dims,nc_
&attr,nc_stat)

 call ncvgt(fileid,so_grps_id,nc_corner,nc_edge,so_grps,nc_stat)
 so_seg_tot_id=ncvid(fileid,'so_seg_tot',nc_stat)
 call ncvinq(fileid,so_seg_tot_id,nc_dummy,nc_type,nc_rank,nc_dims,
&nc_attr,nc_stat)

      ⋮            (233 lines total)

 call ncvgt(fileid,source_segment_ptr_alias_id,nc_corner,nc_edge,so
&urce_segment_ptr_alias,nc_stat)


 call ncclos(fileid,nc_stat)
 end if
```

Finally, the MPI broadcast macros:

```
      call MPI_bcast(bk_num,1,MPI_INTEGER,mpi_root,MPI_COMM_WORLD,mpi_er
     &r)
      call MPI_bcast(background_coords,1,MPI_INTEGER,mpi_root,MPI_COMM_W
     &ORLD,mpi_err)
      if(mpi_rank.NE.mpi_root)then
      ptr_background_n=mem_alloc((((bk_num)-(1)+1)*((zone_type_num(2))-
(
     &1)+1)))
      endif
      call MPI_bcast(background_n,(((bk_num)-(1)+1)*((zone_type_num(2))-

     &(1)+1)),MPI_DOUBLE_PRECISION,mpi_root,MPI_COMM_WORLD,mpi_err)
      if(mpi_rank.NE.mpi_root)then
      ptr_background_v=mem_alloc((((3)-(1)+1)*((bk_num)-(1)+1)*((zone_ty
     &pe_num(2))-(1)+1)))
      endif
      call MPI_bcast(background_v,(((3)-(1)+1)*((bk_num)-(1)+1)*((zone_t
     &ype_num(2))-(1)+1)),MPI_DOUBLE_PRECISION,mpi_root,MPI_COMM_WORLD,m
     &pi_err)
      if(mpi_rank.NE.mpi_root)then
      ptr_background_temp=mem_alloc((((bk_num)-(1)+1)*((zone_type_num(2)
     &)-(1)+1)))
      endif
      call MPI_bcast(background_temp,(((bk_num)-(1)+1)*((zone_type_num(2
     &))-(1)+1)),MPI_DOUBLE_PRECISION,mpi_root,MPI_COMM_WORLD,mpi_err)


      call MPI_bcast(so_grps,1,MPI_INTEGER,mpi_root,MPI_COMM_WORLD,mpi_e
     &rr)
      call MPI_bcast(so_seg_tot,1,MPI_INTEGER,mpi_root,MPI_COMM_WORLD,mp
     &i_err)

           ⋮                    (98 lines total)

      call MPI_bcast(source_segment_ptr_alias,(((so_seg_tot)-(1)+1)),MPI
     &_INTEGER,mpi_root,MPI_COMM_WORLD,mpi_err)


      return
      end
```

This subroutine started out as 39 lines of FWEB code, and expanded to 646 lines of compilable FORTRAN. The implications for code maintenance should be obvious.