

This is an appendix of, *Nonlinear Resonance in the Hydrogen Atom*, Ph.D. dissertation at the University Of Texas at Austin by Marshall Burns, May, 1991. A précis of the dissertation is online at <http://www.MarshallBurns.net/physics/ThesisPrecis.asp>.

---

## E.2 Customizing C for scientific computing

C is the language of choice for doing state of the art computer programming. The features of C allow for the generation of the fastest, tightest executables possible without writing directly in assembly code. It also allows for the manipulation of the execution environment, giving the programmer more control over the run-time process than can be achieved with most other high-level languages.

Usually, C is seen as a “programmer’s language,” one that nonprofessionals are best to stay away from. This is because its speed and versatility come from exposing the programmer to the raw processing power of the computer, which can be difficult to manage. However, learning how to tame that power can be rewarded with professional quality programs, and it’s also fun!

I learned C for the purpose of doing the numerical simulations that form the main body of this work in Chapters 2, 3 and 4. In the meantime, I have acquired several years of experience at writing programs in C. During this time, I have developed a library of header files and utility functions that make my programming task easier. The most important concepts are contained in the header files; they are the subject of this section.

One of the best features of C is the *preprocessor*. This is a convenient tool for the redefinition of words used in programs. It allows the programmer to customize the language to his or her own specifications. This is the main purpose of the system of header files described here.

The program that performed the quantum mechanical simulations in Chapter 4 was written on a DOS computer in MicroSoft C, but the major calculations were performed on a Cray in Cray Standard C. So one of the challenges that the header library tackles is to establish portability between these two computers and compilers. In other words, the goal is to allow for code which can be compiled on either type of machine without modification.

This is not a tutorial in C; a working knowledge of the language is assumed. For programmers who are considering learning C, I recommend starting with MicroSoft Quick C, which not only provides a slick programming environment, but also comes with a tutorial, *C For Yourself*, which takes you through all elements of the language. Once you become familiar with the language, the reference of choice is *C: A Reference Manual* by Harbison and Steele. A little book that I found to be gem of wisdom, that seems to come from a lifetime of programming experience, is *The Elements of Programming Style* by Kernighan. For issues of scientific computing, there is no substitute for the amazingly comprehensive *Numerical Recipes* [87aWHP] series by Press, Flannery, Teukolsky and Vetterling.

### *Custom headers*

The C language proper consists of only 35 words, such as *if*, *int*, and *sizeof*. The main functionality of a commercial C package is in the library of functions that come with it. The core of this library will probably conform to some standard, such as the ANSI standard, for arguments and returned values. In addition, it will probably also include functions for nonstandard operations, such as graphics.

In the documentation of the library functions, you will probably be told that to use a particular function you must include a particular header file. The primary purpose of the header file is to supply a *function prototype* that gives the compiler the pattern of arguments for this function. (Alternatively, if the function is implemented as a macro, then instead of a prototype, the header file will contain the code of the macro itself.) The header may also establish new data types used by the function, and it may define aliases for constants to be passed to the function as arguments.

While it is compulsory to provide the compiler with certain information in the header files, it is not necessary to use the supplied header files themselves. What is described here is a system of header files that I included in my programs instead of those supplied with either MicroSoft C or Cray Standard C. These headers customize the programming environment, while at the same time providing either compiler with the information it needs to run its library functions.

### Avoiding the headers headache

In the standard procedure for writing in C, a source file starts out with a series of include commands for the headers needed. Which headers are included depends on what functions the source file calls. Every time you add a function call, you have to think about whether the appropriate header is already included. If you remove a function call, you might want to pare the include list by removing its header, but you can only do this if that header is not required by another function called in the same file.

Instead of worrying about which headers to include in the source files, it is easier to have *one* set of header files that is included in *all* source files. The following set of files, which are described individually below, provides all the functionality needed for any C program, as well as the customization mentioned above:

- keywords.h
- limits.h and float.h
- constant.h
- keystrok.h
- types.h
- variable.h
- function.h
- macros.h
- prototyp.h

Rather than including these files individually, and in order to reserve the freedom to reorganize the files, the include commands for these files are kept in a master header file. It is *that* file that is included in all of my source files.

### Making C pretty: keywords.h

Personally, I like my programs to read as much like English as possible. The following list of preprocessor definitions allows a dramatic improvement in the appearance of my C code:

```
#define if          if(
#define then       )
#define is         ==
#define isnt       !=
#define set
#define to         =

#define not        !
#define and        &&
#define or         ||
#define NOT        ~
#define AND        &
#define OR         |
#define xOR        ^
#define mod        %
```

This allows the incomprehensible C gibberish:

```
if (a==0 || b!=(q%5)) x=y;
```

to be rendered as the more friendly:

```
if a is 0 or b isnt (q mod 5) then set x to y;
```

The latter version may compile a microsecond or so slower because it requires the compiler to look up several definitions. However, the resulting object code for both versions will be absolutely identical. There is no run-time disadvantage to writing C in this way.

One of the advantages of this set of symbols is that it avoids the common error of testing equality with the assignment operator, i.e., of writing “=” in place of “==” by mistake.

Aside from vocabulary, it is also easier to understand a program if you can see it all on the screen at once. I always try to write programs so that the code for each function takes up no more than one screen. There is one keyword, *unsigned*, that takes up far too much space on the screen for its significance, and

tends to appear frequently. The following definitions keep declarations of unsigned integers from monopolizing the screen:

```
#define CharU      unsigned char
#define ShortU     unsigned short
#define IntU       unsigned int
#define LongU      unsigned long
```

Finally, I begin in this file a practice carried out at great length in the file `function.h`, described below. When a word is formed from two or more English words, I like to make the result easier to read by capitalizing the component words:

```
#define GoTo      goto
#define SizeOf    sizeof
#define TypeDef   typedef
```

Note that all of these definitions serve only to give me the freedom to use the new forms. The original symbols, such as “`goto`” and “`==`” are still in service. It’s just that I choose never to use them directly.

### Defining numeric data types: `limits.h` and `float.h`

These are the only *noncustom* files included in the header list. They are ANSI standard files provided with every compiler to define the numeric data types.

### Machine-specific constants: `constant.h`

Here are defined constants that are used elsewhere to determine whether the program is running in DOS on an IBM compatible or in Unix on a Cray. The list of constants and tests could be expanded almost without limit to allow for other compiling and operating environments.

```
#define Intel8086Family      1
#define CrayMachine          2
#define MS_DOS                1
#define Unix                  2
#define MicroSoftCompiler    1
#define CrayCompiler          2

#if defined(M_I86)
    #define CPU                Intel8086Family
    #define OperatingSystem    MS_DOS
    #define compiler           MicroSoftCompiler
#elif defined(_CRAY)
    #define CPU                CrayMachine
    #if defined(_UNICOS)
        #define OperatingSystem Unix
    #endif
    #define compiler           CrayCompiler
#endif

#if not (defined(CPU) and defined(OperatingSystem))
    #error "The CPU, operating system and compiler have not all been identified."
#endif
```

The constants `M_I86`, `_CRAY` and `_UNICOS` are called predefined macros, and are provided by the respective compilers.

If the machine in use is of the DOS type, then it has segmented memory addresses, and certain code and data will need to be defined as *near* or *far*. If the machine is a Cray, then any appearance of these special keywords will be ignored:

```

#if CPU is Intel8086Family and compiler is MicroSoftCompiler and not defined(NO_EXT_KEYS)
    #define near near
    #define far    far
#else
    #define near
    #define far
#endif

```

The all-important null pointer is:

```
#define null        ((void *)0)
```

This definition appears in many of the standard header files because it is needed by many different types of function. Those files have to provide a test to see if some other file has already made the definition, but that is not necessary here because this file is only included once per source file. Those files also generally follow the convention of naming constants in all-capitals, and call this “NULL.” There is no danger in breaking that convention, as long as all your source files use the name given in the definition. If you wanted to be able to also compile programs written in the standard convention, you could add “#define NULL null.”

Next this file defines the color values and color indices specific to DOS machines:

```

#define cvBlack      0L
#define cvBlue       0x2A0000L
etc.

#define ciBlack      0
#define ciBlue       1
etc.

```

Not all vocabulary has to have unique meanings. Here is a definition of a pair of synonyms for the two Boolean constants, *true* and *false*, which are themselves defined later in the file types.h.

```

#define yes          true
#define no           false

```

This file also contains important physical constants, such as:

```

#define pi           3.14159265358979323846264338327950288
#define c            (2.99792458E10)
#define hBar        (1.0545887E-27)

```

If you decide to use the second definition in your own header file, be careful not to give any variables in your programs the name “c.”

Finally, this file defines the often-used constants,  $2\pi$  and  $\pi/2$ . However, the preprocessor is not equipped to do floating point arithmetic, so they are defined as variables. Alternatively, you could do the arithmetic yourself and enter the result here in a #define command. I prefer to let the computer do the arithmetic:

```

#if ThisIsTheMainSourceFile
    double TwoPi to (2. * pi);
    double PiBy2 to (pi / 2.);
#else
    extern double TwoPi;
    extern double PiBy2;
#endif

```

To accommodate this code, and some other similar examples below, the main source file of all the files making up the program includes the command “#define ThisIsTheMainSourceFile 1.”

In the definition of TwoPi and PiBy2, note the use of the customized assignment operator, “to” for “=”.

### Finding the right keys: keystrok.h

This file contains definitions designed to make it easy to refer to operator keystrokes other than those represented by the usual set of printable characters. This includes both the standard ASCII keys, such as:

```

#define kCtrlG      0x07
#define kRubout 0x08
#define kTab       0x09
#define kCtrlEnter 0x0a
etc.

```

and special keys specific to DOS:

```

#if OperatingSystem is MS_DOS
    #define kHome      0x4700
    #define kUp       0x4800
    #define kPgUp     0x4900
    #define kLeft     0x4b00
    etc.
#endif

```

Code intended for execution on a DOS machine can refer to both sets. Code to be executed on the Cray must only refer to the first.

### Setting up data types: types.h

This file contains all the type definitions. First, we need Booleans:

```
TypeDef enum {false, true} Boole;
```

Next, several kinds of pairs are often useful, such as a pair of bytes, a pair of “shorts,” and two precisions of complex numbers:

```

TypeDef ShortU      PairB;
TypeDef struct {short  e1, e2;} PairS;
TypeDef struct {float  Re, Im;} ComplexF;
TypeDef struct {double Re, Im;} ComplexD;

```

A very important structure type is the one that holds data on an open stream, such as a file. The ANSI standard calls this “FILE,” but I prefer “StreamType” because that is what it is:

```

TypeDef struct
    #if compiler is MicroSoftCompiler
        {CharU *pointer; int count; CharU *base, flags, handle;}
    #elif compiler is CrayCompiler
        {int count; CharU *pointer, *base; ShortU flags; CharU handle; long _ftoff;}
    #else
        #error "Sorry, but I don't recognize the compiler."
    #endif
    StreamType;

```

Note the use of conditional compilation to account for the fact that Cray and MicroSoft define their stream structures differently. In this code, you are free to give the structure elements any names you like, but they must be listed in the order shown and have the individual types shown because already-compiled code in the Cray and MicroSoft libraries will be looking for structure elements of those types at those positions.

Finally, this file defines a special structure that is useful for holding data on the environment and other globally useful information:

```

TypeDef struct
    {Boole initialized; char calls[150]; StreamType *boss, *read, *write;
    Boole NewKeyboard; VideoType video; short pause, beep, verbosity, skip;
    } EnvType;

#if ThisIsTheMainSourceFile
    EnvType EnvData to {0};
#else
    extern EnvType EnvData;
#endif

```

The use of each of the elements of EnvData is quite involved, and I will not go into that here. The point of introducing this object is that it helps to have a place somewhere in memory where useful data can be

accessed and updated without having to pass the address of that location around as an argument to every function. It is best to have all of this data in one structure because that promotes a more organized style of programming than if each of these elements were an independent global variable.

### Predefined global variables: `variable.h`

C has two standard global variables. “`errno`” is a standard name which will be known to the linker, so it must be used. However, it is perfectly healthy to also define and use a synonym:

```
#define ErrNo      errno
extern int near   ErrNo;
```

Both MicroSoft and Cray use an input/output buffer array called “`_iob`”. This is not part of the ANSI standard, so other compilers may require special handling in defining these special streams:

```
#define keyboard   (&_iob[0])    /* stdin */
#define display    (&_iob[1])    /* stdout */
#define ErrorOutput (&_iob[2])  /* stderr */
#define AuxDevice  (&_iob[3])    /* stdaux */
#define printer    (&_iob[4])    /* stdprn */

extern StreamType near _iob[];
```

The names in comments on the right are the ANSI-styled symbols for these streams. The symbol “`_iob`” must be used here because it will be known to both the MicroSoft and Cray linkers. Elsewhere, the streams may be called by the names given here, “`keyboard`,” “`display`,” etc.

Note the use of the special keyword “`near`” in the declarations of both externals, `ErrNo` and `_iob`. This keyword is defined to be ignored by the Cray compiler. (See `constant.h`, above.)

### Making function names more friendly: `function.h`

This is the simplest header file. All it contains is aliases for functions whose given names I don’t like. Some functions are renamed just to capitalize component words that go to make up the names. Others are given new names that seem to better describe what they do. For example:

```
#define CAbs      cabs
#define CAlloc    calloc
#define CPUTime   clock
#define execute   execl
#define FClose    fclose
#define FPrintf   fprintf
#define FractionPart modf
#define GetPID    getpid
#define InvSin    asin
#define LocalTime localtime
#define OutByte   outp
#define remainder fmod
#define SqRoot    sqrt
#define StrNCmpstrncmp
#define StrToDbl strtod
#define StrToInt  atoi
```

### Where the work gets done: `macros.h`

The wonderful thing about the `#define` command is that you can use it for the most trivial purpose of changing the name of a function, or you can use it define very sophisticated macros. Below are examples of some macros that I have found particularly useful.

#### *Conditional debugging*

C is famous for its speed and versatility, but those features come at the cost of exposure to dangers that other languages shield the programmer from. For example, there is no check for arithmetic overflow, so incrementing an integer too many times can lead to unpredictable results. Arrays have no defined boundaries, so errant data operations can modify the status of unexpected memory locations. The

responsibility falls upon the programmer to anticipate and handle overflows and to perform array operations within the correct bounds.

Writing in C does not mean that programs cannot have error checking. It means that the decision of where and when to check for errors is left to the discretion of the programmer. The optimal way to write correct code that functions as intended and yet does not waste time checking for errors that cannot happen, is to implement a two step strategy:

- Apply stringent error checking while writing and debugging the program.
- When the program has been fully tested, reduce the checking to just handle operator error.

This variable level of error checking can be implemented by combining the preprocessor's features of macro definition and conditional compilation.

For example, it is a good idea to check all of the arguments on entry into a function, to ensure that they have values that fall within expected ranges, and to make sure that pointers are not null. Another good idea is for every *switch* statement that is expected to route through one of its *case*'s to end with a *default* statement that flags an error. It is also good practice to check the value of `ErrNo` every so often to see if an unexpected error has occurred<sup>1</sup>. However, all of these procedures will slow a program down. That is where conditional compilation comes in.

The following are examples of error-checking macros. They use a messaging system called *error* which is not described here. `CheckForError` issues a message if `ErrNo` is nonzero. `CheckForNull` prints a message if its argument is a null pointer (or a zero integer). `CheckBoolean` and `CheckRange` complain if their respective first arguments do not have appropriate values. And `CaseError` is designed to be called by a *default* statement that should never be executed.

The important thing about these macros is that they only exist if the symbol *debugging* is defined and nonzero. If *debugging* is undefined then the compiler passes over every occurrence of these macros without generating any object code. This means that you can install these macros throughout your program without concern for degrading its ultimate performance.

Once a program is debugged, it is better to turn the macros off by not defining *debugging* than to remove them from the source file. Not only is it easier, but also, when you come back to the program later to make modifications you will want to debug it again, and the macros will be sitting there ready to go to work.

```
#if debugging
    #define CheckForError(label) if ErrNo then error("ErrNo", label, ErrNo); else
    #define CheckForNull(pointer) \
        if pointer is null then error("null", #pointer); else
    #define CheckBoolean(Boolean) \
        if not IsEither(Boolean, 0, 1) then \
            error("I have '%s' = %i when it should be either 0 or 1.", #Boolean, Boolean); \
        else
    #define CheckRange(value, low, high) \
        if not IsOrdered(low, value, high) then \
            error("range", #value, (double)(value), (double)(low), (double)(high)); \
        else
    #define CaseError(variable) error("case", #variable, variable, variable)
#else
    #define CheckForError(label)
    #define CheckForNull(pointer)
    #define CheckBoolean(Boolean)
    #define CheckRange(value, low, high)
    #define CaseError(variable) error("case: " #variable)
#endif
```

Here is another example of conditional debugging. Infinite loops are particularly frustrating for two reasons. First, the only way to break into one (unless you've left open an interrupt mechanism) is to turn off the computer. But worse, if a loop is not generating some output, then getting stuck in it is often

<sup>1</sup> Surprisingly, MicroSoft does not anticipate this practice. At the startup of any executable, the value of `ErrNo` is unpredictable [Paul E., MicroSoft technical support, 89 08 15].

indistinguishable from having the computer hang for any other reason, so they can be hard to diagnose. The following code is designed as a replacement for the *for* statement in C, to prevent infinite loops.

```
#if debugging
    #define loop(LoopInitializer, LoopWhile, LoopIterate, LoopAction) \
        {double LoopCount to 0., LoopQuit to 1E5; \
          for ((LoopInitializer); (LoopWhile); (LoopIterate), LoopCount++) \
            {LoopAction; \
              if LoopCount > LoopQuit then \
                {if error("loop",LoopCount,#LoopInitializer,#LoopWhile,#LoopIterate) \
                  is 'R' then break; else LoopCount to 0.;} \
            } \
        }
#else
    #if ThisIsTheMainSourceFile
        double LoopQuit;
    #else
        extern double LoopQuit;
    #endif
    #define loop(LoopInitializer, LoopWhile, LoopIterate, LoopAction) \
        for((LoopInitializer); (LoopWhile); (LoopIterate)) {LoopAction;}
#endif
```

The first three arguments are expressions, and serve the role of the three expressions appearing in a *for* statement. The last argument is a statement, and serves the role of the body of the *for*. A semicolon is optional on the statement in the last argument, and on the *loop* macro itself. Compound expressions in the first three arguments must be enclosed in round brackets. Braces on a compound statement in the last argument are optional. The arguments are separated by commas, not by semicolons as in *for*.

The difference between *loop* and *for* is that when *debugging* is defined and nonzero, *loop* keeps track of how many times it has iterated, and issues a warning if it goes on for too long. “Too long” means, by default, more than 100,000 times; this can be reset by assigning a different value to *LoopQuit* in *LoopInitializer*, as in:

```
loop((x to 2, LoopQuit to 1E9), x < 1E6, x * x, null);
```

If the warning is issued, the loop can be continued by pressing “r” (lower case), or exited by pressing “R” (capital). (This is because of the design of *error*, which allows processing to Resume if “R” (either case) is pressed, and otherwise terminates processing.)

If *debugging* is not defined, then *loop* is exactly identical to *for*. There is no efficiency cost, in terms of either speed or size, for using *loop* with *debugging* off. The reason *LoopQuit* is defined as a global when *debugging* is off is so the program will not choke on code such as the example above that sets an alternate value for *LoopQuit*. If *debugging* is off, then changing the size of *LoopQuit* has no effect.

The final example of conditional debugging demonstrates checking the return value of a function and issuing a message (through *error*) if the value is not the expected one. But this particular example also perform another operation, which I encourage you to make note of if you are just getting started in C, because I found this to be a very sly and destructive source of error. If your program writes to a null pointer and you don’t know about it (which can happen if the write is unintended, rather than just to a bad address) the effect is a “null pointer assignment.” (*fwrite* returns the correct count.) In Microsoft C, this will cause a run-time error message *after program termination*, and *only if the program is compiled with pointer checking on*. Otherwise the error will go undetected, and will have unpredictable effects. If the message is given, it will come without any clue as to when or where or how the bad assignment took place. Null pointer assignment bugs are very hard to find. The following code is designed to catch one common cause of them:

```
#if debugging
    #define put(stream, source, size, count) \
        {CheckForNull(stream); \
          if fwrite(source, size, count, stream) isnt count then \
            error("put", stream, #stream, count, size, #source, #stream);}
#else
    #define put(stream, source, size, count) fwrite(source, size, count, stream)
#endif
```

*Logic and arithmetic*

Here are three simple macros that are useful under various circumstances:

```
#define IsEither(a, b, c)      ((a) is (b) or (a) is (c))
#define IsBoth(a, b, c)      ((a) is (b) and (a) is (c))
#define until(expression)   while (not (expression))
```

The first two work only with integers or pointers. Generally, equality of floating point numbers cannot be safely tested with *is* (`==`).

Many simple but important arithmetic operations are best performed with macros. Here are some examples:

```
#if compiler is MicroSoftCompiler
    #define mag(x)      ((double)(x) > 0. ? (x) : -(x))
#else
    #define mag(x)      ((x) > 0 ? (x) : -(x))
#endif
#define max(a, b)      ((a) > (b) ? (a) : (b))
#define min(a, b)      ((a) < (b) ? (a) : (b))
#define up(a, b)       if (b) > (a) then (a) to (b); else
#define down(a, b)     if (b) < (a) then (a) to (b); else
#define power(pwr, base, n) \
    {short MCRi; pwr to base; for (MCRi to 2; MCRi <= n; MCRi++) pwr *= base;}
```

The important considerations to keep in mind are:

- Macros are faster than function calls, but complex ones will increase the size of a program.
- Beware of unwanted side effects. For example, `mag(x++)` will increment `x` twice.

There are some useful tricks for manipulating complex numbers and matrices in C. I learned most of mine from *Numerical Recipes in C* [87aWHP, Appendices D, E], to which I defer this discussion on those subjects.

*Stream operations*

Reading and writing to and from files can often be simplified using macros. The following examples use the macro `put` defined above under “Conditional debugging.”

```
#define PutItem(stream, item)    put(stream, &(item), SizeOf(item), 1)
#define PutItems(stream, item, count)    put(stream, &(item), SizeOf(item), count)
#define PutList(stream, item, count)    put(stream, &(item), SizeOf(item)*count, 1)
#define PutString(stream, string)    put(stream, string, StrLen(string), 1)
#define PutAt(stream, position, source, size, count) \
    {seek(stream, position); put(stream, source, size, count);}
```

**The quintessential header file: `prototyp.h`**

This file contains what make up the main contents of most of the standard header files, that is, the function prototypes. Included are prototypes for all of the functions that are ever used in any of my programs, so it serves universally, although personally. If at some point I write my own function or macro to replace a standard library function, then I remove the prototype from this file. That way, if I inadvertently call the standard function, the compiler will warn me that there is no prototype for it.

**Beyond headers**

That completes the description of the header library. But it is often useful to define macros in regular source files as well.

*Collapsing code*

Any body of code that repeats the same sequence of steps can be collapsed into a series of macro calls. If the sequence of steps is fairly long and complicated, then it is better to define it as a function rather than a macro. But for fairly simple but repetitive operations, macros are very handy.

The following example is used in a program that allows the operator to change the values of data items on the screen. Any one item can be changed before moving on to the next, but the operator can move

around the screen at will before changing anything. The program looks for a flag that indicates that *something* has changed. Then it looks through all the fields on the screen to see which one is current, and changes the data for that item. The search through the fields is coded as:

```
if      window->field is item1 then data->item1 to *(double *)item1->value;
      else if window->field is item2 then data->item2 to *(double *)item2->value;
      else if window->field is item3 then data->item3 to *(double *)item3->value;
      else . . .
```

where the *itemn*'s are the names of fields on the screen as well as elements of the structure *data*, and *window->field* is the current field. This long list of repetitive code can be replaced by:

```
#define update(item)      \
      if window->field is item then data->item to *(double *)item->value
update(item1); else update(item2); else update(item3); else . . .
#undef update
```

Collapsing code in this way can go a long way to making programs easier to understand. But be careful about indiscriminate use of macros. Remember that every macro call expands into the full text of the macro before it is compiled. So if your macros are very long and complicated, or if you often have several layers of nested macro calls, then a segment of code that appears on the screen as very short may compile into a lot of object code.