A legacy note for a Prolog newsletter on rapid prototyping of graphics using Visual Prolog

Scouring the bulletins boards around Toronto, I've picked up messages from a few people who, like me, do a great deal of prototyping and thought that Prolog seemed like good language for this purpose. A few were people like myself, who use their computers to make art. One of the peculiarities of such use of computers is that you use many programmes that you write only once, or a few times, to do something a bit unusual, you hope, and then never use them again. When you are using a programme once, or a few times, you much more interested in the time it takes to write the code than the time it takes to run. If it takes a few minutes extra to run but several hours fewer to write, who cares? The net-saving in time, if you use the programme only once, is still a few hours; the saving compensates for the displeasure you experience when watching the thing churn away at a speed you know you could increase if you were to sink another few late nights into it.

Several messages I picked on the bulletin boards were written by people who had read about PDC Prolog, formerly Turbo Prolog, from Prolog Development Center. They found they needed to link 'C' modules with Prolog modules, and expressed annoyance with difficulties in doing so. I'd like to explain how one does this--and it really isn't difficult. Then I'd like to present a few graphics programmes, as examples that show where Prolog really shines. I hope that few non-Prologists might compare the Prolog code with 'C' code that would accomplish the same result, and marvel a little at its elegance. I believe that, chosing Prolog makes excellent sense for many applications and I hope that I might persuade a few programmers to try it out. After all, if you can call on the 'C' code that you've written, what do you have to lose?

As an example to illustrate how to link 'C' and PDC Prolog, I've chosen to work with the Fastgraph library from Ted Gruber software--an excellent graphics package, though in need of being updated to offer VESA suppor, now that many graphics programmers are running SVGA boards. I do a fair bit of graphics programming, and I have always been able to do what I wanted to easily using Fastgraph, provided I accept the standard VGA modes. I have chosen Gruber's Fastgraph library for my illustrations, partly because of its quality, partly because it is available on many bulletin-boards in a shareware version (that lacks support for world-space coordinates and displays a nag screen whenever you initialize the graphics, but is a full working version otherwise identical to the registered version of the library) and partly because graphics applications are so enjoyable, yet are the sort of application to which Prolog is rumoured to be unsuited. As importantly, the problems that one encounters with this library are typical of problems that we encounter in creating an interface between PDC Prolog and library of functions written in 'C'--if we can handle this one, we can handle just about any library. Because the shareware version Fastgraph/Light is available freely, anyone can test how easy it really is to link 'C' code and PDC Prolog. Fastgraph/Light is available on mainy bulletin boards; if you cannot get it locally, you can download it from the DustDevil Bulletin Board in Los Vegas, NV ((702) 796-7134)).

For those who, like myself, remember the bad old days of personal computing when you'd risk $100 or $150 on a product only to find out it didn't do what you hoped it would, the chance to try out the product and to see that satisfies your needs is wonderful. I tested out the procedures and methods for calling 'C' Fastgraph routines from Prolog before giving Mr. Gruber a cent, so when I did register, I knew that what so many people had claimed on the bulletin boards was so difficult, could be done, and done easily. Gruber provides support for BASIC, Pascal, and 'C', but, understandably, not Prolog. Everyone knows that attempting to do graphics programming without a decent graphics library is doing much needless work. However, there are no graphics libraries written in Prolog. What do you do in this situation?

Call the appropriate 'C' code. This is what you are doing when you run PDC Prolog anyway, so you can be sure that it works well. PDC Prolog is written in 'C', and a major component of the package you purchase when you buy PDC Prolog is a file named 'prolog.lib . The library is simply a set of 'C' functions that the Prolog compiler links into your '.exe' when your programme code calls them. Because they are simply a set a 'C' functions in '.lib' form, you can, as an extra benefit, call any of these routines from 'C', as you would any other 'C' library, and so take advantage of their excellent windowing system, etc. But we are interested not in how you call PDC Prolog routines from 'C', cut how you call a 'C' library from PDC Prolog. You can do pretty much the same way you invoke the 'prolog.lib' routines--that is, almost invisibly. When using 'C' libraries other than 'prolog.lib' there are two small differences. First, you have to tell your compiler not to give up if it doesn't find the code for a few procedures when it is building the executable file [the '.exe' file.] file--that code will come along later when you link the graphics library, or whatever, into the '.exe' file. You do this in much the same way you do it 'C'-- you tell the compiler that the code for some functions (in Prolog we call them predicates) comes from an external file. We don't do this by labelling them 'extern' as the C-people do; we do it by flagging the predicates as "global."

There are a couple restrictions on this; but, thank Goodness, they're not inconvenient. One is that you must declare all global predicates before declaring any non-global predicates. The other is that, if you work in the Integrated Developers Environment and link modules with a '.prj' file (done in much the same way as it was done in Turbo C 2.0, essentially by listing the files that needed linking, one file name per line), all the modules must have the same set of global predicates. There is simple and neat way to meet both requirements, one 'C' programmers are familiar with it--you write a header file. Customarily, these files have a '.pre' file extension, instead of the '.h' that C-people use--the '.pre' extension being a mnemonic for their being files of predicate declarations. While the '.h' files that 'C' programmers write typically have several statements with the same form as the following:

```
extern void rect(int,int,int,int);
extern int  putpixel(int,int,int);
```

a comparable '.pre' file contains the heading

GLOBAL PREDICATES

The "**GLOBAL**" qualifier in the ".pre" file does the same job as "**extern**" in the ".h" file that C-people use. Then, under the heading, there are statements with the same form as:

rect(integer,integer,integer,integer) - (i,i,i,i)
putpixel(integer,integer,integer,integer) - (i,i,i,o)

These declarations bring us to one of the differences between standard Prologs (sometimes called "Clocksin & Mellish" Prolog after the authors of the book *Programming in Prolog*) and PDC Prolog. Unlike standard Prolog, PDC Prolog is a typed language, and the types for all the variables in a predicate's parameter list must be specified in the predicate declaration (which occurs in the "**PREDICATES**" or "**GLOBAL PREDICATES**" section of a programme. This difference caused considerable consternation in the Prolog programming community when Turbo Prolog first appeared. There are obvious advantages to it, that make PDC Prolog a better general-purpose language. For one thing, it provides for better error checking. More importantly, it makes for greater efficiency. But the consternation was not well-founded anyway, for PDC prolog allows user-defined domains, and one can define a domain, call it an anytype_domain, whose "functor"--a token that identifies a syntactical structure as a

user-defined domain and that ties together the varibles in the domain into a single object. We can other special function to model any data type whatsoever and then declare all predicates in a programme as taking “any_type_domain”. (If this sounds horrible, it is, but it is actually what standard Prolog’s do.) The *PDC Prolog User’s Guide* shows you how to do this in Appendix I, devoted to metaprogamming.

The other feature that PDC Prolog lacks that standard Prolog’s have is the ability assert and retract rules at run-time. This difference caused even greater consternation than the use of type-checking, and again, the consternation was not well-founded, firstly, because PDC Prolog supplies an interpreter than has this capability, and secondly, because even this capability can be modeled in PDC Prolog--Safaa H. Hashim and Philip Seyer’s, *Turbo Prolog: Advanced Programming Techniques* shows you how to do this. But the lack does not seriously limit PDC Prolog as a general-purpose langauge; metalogical capabilities are needed in special-purpose applications, primarily in artificial intelligence, in creating systems that do “nonmonotonic reasoning.” Ordinarily, it is not something one wants to get involved with, since having, and using, the ability to assert and retract rules at run-time means that the programme alters its own code while it is running. This makes debugging a bit difficult.

PDC Prolog demands that all the global predicates be specified before any non-global predicates. A consequence of this requirement is that global and non-global predicates cannot mix with each other in the code. Furthermore, all the files in a PDC Prolog project must include the same list of global predicates. In practice, these two requirements mean that when you are working on a large, multi-file project, you create a file that holds all your globals--your global predicates and global constants--and “include” it at the beginning of your list of “include”’s. When you finish writing the code for one module in your project, you pull out the type specifications for those predicates that you want to call from another module, transfer them to the “.pre” file, and append a “-“ and the information in parentheses after the dash.

Though I have not made any use of “**project**”s in the examples below, a word on ‘.prj’ files is in order, since they have, too, have an undeservedly bad reputation that, perhaps, hinters some programmers from making using PDC Prolog for large, multi-file tasks. Mick McAllister’s otherwise flawless book, *Illustrated Turbo Prolog 2.0*, for example, suggests that it the author’s intention to “talk you out of using the project concept in your programming.” (p. 290) He quite correctly points out that almost none of the Turbo (PDC) Prolog primers include any discussion of the use of projects, and that “none of the programmes included with your original Turbo Prolog disks are projects” (*loc.cit*)--a situation that hasn’t changed with more recent releases of from the Prolog Development Center. Be that as it may, projects are not difficult to handle. The following is example of a project file for another project I was working on--an set of routines for algorithmic composition.

```
% algi.prj  == not this line must be omitted when using
% a project all that can occur in the project is a
% series of names of files.  Normally all these files will
% have “.pro” as their extension, but  what the linker will
% look for, really, is files with the % extension ‘.obj.’
% The extension must not be specified.

pitchnms+
periods+
multiper+
col_rhyt+
microrhyth+
rhythfam+
prmlists+
```

scales+
intervals+
goalfile

The first file, 'pitchnms.pro' begins as follows:

```
/* pitchnms.pro */
/* file number 5 in the algorithmic composer project */

project "algi"
include "b:\\composer\\msupport.pro"
```

Every file in the list of files named in the '.prj' file must include the statement "**project "<project name>"** " as the first line. The second line here is a standard "**include**" statement. I like to include all my utility predicates--predicates for doing such things as appending an item to list, or for adding up a list, or for deciding whether an item is a member of list, or my "**close_enough_equals()**" for reals, etc., in a single file. I include that file as in every single one of the files in the project (all the files named in the list of files in '.prj' module), and then, in that file, I include other files that create user-defined domains and declare the global predicates that the project uses. Thus the first lines in 'msupport.pro' (a mnemonic for 'music support') are

```
/* msupport.pro */
/* file No 4 in the algorithmic composer project, "algi" */
/* various support facilities for the algorithmic composer
 project */

include "b:\\composer\\mdoms.pro"
include "b:\\composer\\mpreds.pro"
```

The file 'mdoms.pro'--mnemonic for 'music domains'--consists of a set of domain declarations:

```
/* MDOMS.PRO  */
/* No. 1 file in the algorithmic composer project, "algi" */
/* This file contains the globals for the algorithmic composer
project */

GLOBAL DOMAINS

integerlist = integer*
list_of_integerlists = integerlist*
reallist = real*
charlist = char*
stringlist = string*
symbollist = symbol*
heavy, light = symbol
resultant = res(integerlist,integerlist)
rhythm_lists = integerlist*
chord = chrd(integerlist)
```

chordlist = chord*

GLOBAL DATABASE - music

```
determ notes_per_octave(integer)
determ scale(integerlist)
nondeterm note(integer,symbol)
determ permlist(list_of_integerlists)
determ current_interval_pattern(integerlist)
```

The use of the qualifier "**determ**" for all predicates, including database predicates, that are determinate is not crucial, and can be ommitted, as "determ" is the default for global predicates is PDC Prolog. However, if there were nondeterminate predicates, the use of the qualifier "nondeterm" would be required.

The second file that 'msupport.pro' "**include**"s is 'mpreds.pro', a file that declares all the global predicates that the project uses. When I am working on a project, I write a module, then consider what predicates I shall want to call from some other module. Those that I shall, I transfer to the 'mpreds.pro' module.

```
/* MPREDS.PRO */
/* file no 2 in the algorithmic composer project */
```

GLOBAL PREDICATES

```
nondeterm notenumber_notename(integer,symbol) - (i,o)
nondeterm notelist_namelist(integerlist,symbollist) - (i,o)
nondeterm notename_notenumber(symbol,integer) - (i,o)
nondeterm namelist_numberlist(symbollist,integerlist) - (i,o)
calculate_resultant(integer,integer,resultant) - (i,i,o)
write_resultant( resultant ) - (i)
calculate_periods_beats(integer,integer,integer,
                        integerlist,integerlist) - (i,i,i,i,o)
calc_spans_and_beatlist(integerlist,integerlist,
                        integerlist) - (i,o,o)
collate_rhythms(integer,rhythm_lists,integerlist) - (i,i,o)
calc_multiresultant(integerlist,resultant) - (i,o)
fill_with_micropattern(integer,integerlist,integerlist)
                        - (i,i,o)
compute_family_of_grade_up_rhythms(integer,integer,
                                   list_of_integerlists)  - (i,i,o)
permute_lists(list_of_integerlists,list_of_integerlists)
                                              - (i,o)
calculate_scalelist(integerlist) - (i)
show_scale(integer,integer) - (i,i)
get_scale(integer,integer,integerlist) - (i,i,o)
calculate_scale_of_repeating_octaves(integerlist) - (i)
calculate_scale_of_repeating_intervals
                        (integerlist,integer) - (i,i)
make_schillinger_scale(integerlist,integer) - (i,i)
show_permlist
get_permlist(list_of_integerlists) - (o)
```

make_first_permrelative_scale(integerlist) - (i)
switch_permrelative_scale
switch_interval_relative_scale(integer) - (i)
construct_chord(integer,integer,integer, chord) - (i,i,i,o)
construct_chordlist_for_octave(integer,integer,chordlist)
- (i,i,o)
make_all_scalechords_for_octave_interval
(integer,integerlist,integer,chordlist) - (i,i,i,o)
write_chord(chord) - (i)
write_chordlist(chordlist) - (i)
reverse_chordlist(chordlist,chordlist) - (i,o)
initmusic
beta_distribution(real,real,integer,reallist) - (i,i,i,o)
bilateral_exponential_distribution
(real,integer,reallist) - (i,i,o)
cauchy_distribution(real,integer,reallist) - (i,i,o)
exponential_distribution(real,integer,reallist) - (i,i,o)
gamma_distribution(real,integer,reallist) - (i,i,o)
gaussian_distribution(real,real,integer,
integer,reallist) - (i,i,i,i,o)
weibull_distribution(real,real,integer,reallist)
- (i,i,i,o)
cosh_distribution(integer,reallist) - (i,o)
descending_order(integer,integer,reallist) - (i,i,o)
ascending_order(integer,integer,reallist) - (i,i,o)
triangularly_ordered_distribution
(real,real,integer,reallist) - (i,i,i,o)
descending_linear_distribution(real,integer,reallist)
- (i,i,o)
ascending_linear_distribution(real,integer,reallist)
- (i,i,o)
triangular_distribution(integer,reallist) - (i,o)
logistic_distribution(real,real,integer,reallist)
- (i,i,i,o)
poisson_distribution(integer,integer,integerlist)
- (i,i,o)
random_random_distribution(integer,integer,
integer,integerlist) - (i,i,i,o)

There is, one can see, no distinction between how calls an global Prolog predicate (an 'extern' predicate, defined in another Prolog module) and a global 'C' predicate (an 'extern' predicate written in a 'C' module). In fact, it is possible to compile the 'C' code into an '.obj' file, and include the name of the 'C' '.obj' file, but without the '.obj' extension, in the list files that appears in the '.prj' file. Since the '.prj' file is used only linking, not compiling, the linker is actually looking for '.obj' modules for the files listed in the '.prj' file anyway.

The following is a portion of the file the states the rules for one of predicates given above, "

/* random.pro */
/* file no 14 in the algorithmic composer project */

```
/* this file includes procedures for calculating
   random distributions of various sorts */

project "algi"

include "b:\\composer\\msupport.pro"

CONSTANTS

pi = 3.14159265

/*-----------------------------------------------------------
  utility predicates: power
-----------------------------------------------------------*/


PREDICATES

power(real,real,real)

CLAUSES

power(Base,Coeff,Power) :-
            bound(Base),
            bound(Coeff),!,
            Power = exp(Coeff*ln(Base)).

power(Base,Coeff,Power) :-
            bound(Base),
            bound(Power),!,
            Coeff = ln(Power)/ln(Base).

power(Base,Coeff,Power) :-
            bound(Coeff),
            bound(Power),!,
            Base = exp(ln(Power)/Coeff).

/*-----------------------------------------------------------
    beta_distribution(Prob0,Prob1,No_to_Output,OutputList)
-----------------------------------------------------------*/


PREDICATES

beta_distribution_aux
          (real,real,integer,integer,reallist,reallist)

CLAUSES

beta_distribution(Prob0,Prob1,No_to_Output,OutputList):-
         beta_distribution_aux
```

```
                (Prob0,Prob1,1,No_to_Output,[],OutputList).

beta_distribution_aux(_,_,Count,MaxCount,OutputList,OutputList):-        Count >
MaxCount, !.

beta_distribution_aux
                (Prob0,Prob1,Count,MaxCount,Accum,OutPutList):-
        Count <= MaxCount,
        InvProb0 = 1 / Prob0,
        InvProb1 = 1 / Prob1,
        random(Ran1), Ran1<>0.0,
        random(Ran2), Ran2<>0.0,
        power(Ran1,InvProb0,Val1),
        power(Ran2,InvProb1,Val2),
        Sum = Val1 + Val2,
        Sum <= 1,
        BetaVal = Val1 / Sum, !,
%       write("\n No", Count),
%       write(" Beta Distribution Value = ",BetaVal),
        NewCount = Count + 1,
        beta_distribution_aux(Prob0,Prob1,
                NewCount,MaxCount,[BetaVal|Accum],OutPutList).
```

The predicate "**power()**" is not defined as a global predicate, since it was written to be used the rule for "**beta_distribution()**", which appears in the same module. The predicate is interesting because it shows how a single operator may be "overloaded"--in this case used with different flow patterns (see below). Note, too, the inclusion of 'msupport.pro', in this file as in every file named in the list of files included in '.prj' file. Because 'msupport.pro' "**include**"s the files 'mdoms.pro' and 'mpreds.pro'--the files that include the declarations for the global domains and the global predicates, all files in the project include the same globals. This is one of conditions that we gave above for the use globals in PDC Prolog.

Alert readers might have noted something a little peculiar, apart from the letters in the bracket, in Prolog type specifications for the predicates that are written in 'C', not in Prolog as the predicates immediately above are. Consider the BGI function "**putpixel()**"--the 'C' function has has three parameters, and the Prolog predicate has four! Some may even have connected the difference to the existence of an 'o' in brackets--the number of 'i's in the parenthesis after the dash in the Prolog declarations always matches the number of parameters in the 'C' function declaration. Why?

The answer points up one of the beauties of programming in Prolog. The designers of 'C' choose to tie the specification of what constitutes a function in 'C' to the mathematical concept of a function; so a function must return a single value (in 'C' a function can lack a return value, i.e., have a return value that belongs to the class void) for any specific set of values in the domain of the function. A consequence of this is, not wholly adequate (but one that Prolog stipulations on predicates does not completely overcome), is that everything the function does when it runs, other than returning a return value, is, strictly speaking, a side-effect. That many functions written in 'C' have no return value is one indication of how often programmers use functions to create side-effects.

However, this is not an area in which Prolog is really much better, for every screen display created using Prolog is, by the same token, a side-effect. Perhaps Prolog even comes off a little worse in this comparison, for Prolog is widely touted as a declarative rather than a

procedural programming language. That is, a Prolog programme ideally is logical specification of the problem to be solved rather than a set of instructions to perform. Each predicate in a Prolog programme ideally can be interpreted as a theorem or an axiom and the rest of the statements that make up the clauses for any predicate, when it is a theorem, consists of series of statements from which the theorem will validly follow. (Axioms, which in Prolog are called facts, of course, are true unconditionally.) In theory, the Prolog interpreter determines by itself, using the mechanism of backtracking, whether it can establish the truth of the each of the statements from which the theorem follows, first attempting one proof procedure, then another, until it runs out of alternatives. The use of side-effects relies on the actual sequence by which Prolog "proves the theorems" it is asked to prove, and thus forgoes this basic advantage which Prolog programmes are said to have. Worse, it runs counter to the ideals of declarative programming.

However, Prolog is not really a declarative programming language. At best, it is only semi-declarative. Prolog should thought of as language whose programmes have "a double aspect"--one can interpret a Prolog programme either declaratively or procedurally. Considered declaratively, a clause sets the conditions that must be proved to establish the truth of a Prolog theorem. The ":-" in a clause (see programmes below) means "if" (as if is used in the predicate logic, namely to state that the antecedent be true and the consequent false), while the commas is the predicate calculus' "and." In most clauses the statements specifying a set of conditions are 'and-ed' together, though Prolog does have symbol, ";" for disjunction ("inclusive-or"); these conditions, known as the "body" of a Prolog clause, constitute the antecedent and the part of the clause that precedes the ":=", known as the "head" of the clause, constitute the consequent of statement in the first-order predicate calculus.

This is one way to interpret a Prolog clause. However, we can also interpret a Prolog clause procedurally. Under the procedural interpretation, a Prolog clause is set of instructions, just as a body of 'C' function is a set of instructions. We can consider the form of a Prolog clause as stipulting that, in order to demonstrate that head of the clause is true, each of the following statements (presuming the statements are "and"ed together, as they usually are) should be proved in turn. Each of the instructions can have side-effects if desired.

The 'C' concept of function is modelled on the mathematical concept of a function, while the Prolog concept of a predicate is modelled on the predicate calculus' concept of a theorem. Like mathemathical functions, 'C' functions are one-to-one relations, i.e., given any set of values in the domain of the function (for a 'C'-function the analogous statement is, 'given and set of values of values for the variables in its formal parameter list), a function must return a single value. The function that most 'C''ś call "atan()" is what trigonometrists call the 'Arctan' function. There are an infinite number of values that, given a value for 'x', satisfy the relation "arctan(x) = y" (i.e., that are the angle whose tangent is equal to 'x'), since, for one thing, given one of the 'y''ś that satisfies every Y = y + n*pi satisfies it whenever 'n' is an integer value. The Arctan function was constructed to make one-to-one relation out of th one-to-many arctan relation, and does so by restricting its range to values the function can take on to angles that lie strictly between the limits -pi/2 and pi/2.
Trigonometrists say that the Arctan function returns the principal value of the artan relation.

One might be a little uncharitable towards 'C' and note that it's odd to see it conforming to such mathematically rigorous requirements on the matter of functions being one-to-one relations when other features of 'C' plays so fast and lose with mathematical rigour, as 'C' does when it forgoes the logical demand for substitutibility, which is a fundamental requirement for any mathematically rigorous system. (A system meets the condition of substituibility if it allows one expression that somewhere has been to be equal to second to be substituted for the second expression wherever it occurs. Substitutability ensures that if, within the proof of a theorem a theorem, we obtain the result that A = B - C, and then later, that D = A + E, we can substitute (B - C) for A in the second equation, to obtain D = B - C + E. But we cannot do this

with lines of code that make up a 'C' function; we cannot substitute B - C for A automatically in any later expression, because we cannot know we cannot know whether, during the run, one or more of the variables, A, B, C, have taken on new values. The variables lack referential transparency, the property that ensures that the values expressions of expression are not context sensitive, that the meaning or value of expression can be understood in isolation from other parts of the code, that we do not need understand the entire operation of the code to understand any particular line of it. 'C' lacks referential transparency because, among reasons, it permits statements like x = x + 1; -- a statement that is mathematically absurd. Here, and in any such destructive assignment, the meaning of the variable x is context-sensitive--it has a different value on either side of the '=' sign, and it is this that prevents us from doing what we should be able to do, to subtract x from either side of the statement. For doing so produces the patent absurdity of 0 = 1. Prolog adheres more strongly to requirement of substitutibility and cleaves more closely to referential transparency. It disallows destructive assignments of the sort I just set out. In Prolog, the code would be NewCount = Count + 1. It is a major reason why Prolog programmes are more limpid than 'C' programmes).

Less uncharitably, we can acknowledge that the purpose of modelling 'C' functions on the mathematical concept of a function and of restricting them to have a single return value clear and that the restrictions gives the language great power. 'C' allows you to use a function almost anywhere you can use an expression, and an and to use an expression nearly anywhere you can use a variable. Hence, you can write things like

```
for(int i= 0; i <  getmaxx(); i ++)
```

without the gruesome encumbrance of having initially declared a variable

```
int maxx;
```

and set

```
maxx = getmaxx();
```

One reason you can do this is that functions and expressions both return pointers to one of the standard types, so functions and expressions can be interchanged easily.

Tying the concept of a function in 'C' to the mathematical concept of a function does create limitations, however. Often we would like to assign a value for several variables, and it is handy to do the chore all at once, using a single function. We can squeeze several assignments out of a function in a couple of ways, of course; one way is to assign the variables a 'file' scope, to write a principal function and several subfunctions that the principal function calls one after another, and to have each subfunction assign a value to one of the variables whose values we want to initialize or alter. More likely though, a 'C' programmer would do just what 'C' programmers love to do, and that is to use pointers.

Prolog is no different in this respect; except Prolog programmers almost never have to think about passing by reference, or dereferencing, or "suspicious pointer conversion" warnings. You can pass a Prolog predicate any number of parameters, and get back any number of parameters. What the Prolog interpreter/compiler--Prolog's usually comes with both an interpreter and compiler, though in this article we are mostly talking about compiled versions of Prolog that can create '.exe' files that incorporate library code--does is to decide which parameters are bound--have a value assigned to them when the function is called, and which take on values as the predicate completes its tasks. Technically, what it needs to do is to set up pointers for those variables that take on values as the predicate completes its tasks.

The reason for difference between the input and output variables relates to how 'C' (PDC

Prolog runs in 'C') handles the two sorts of varibles. Whenever a new procedure is begun, a stack frame corresponding to the procedure is created. The stack frame consists first of return pointer, the address the instruction point returns to when the procedure has been completed. It also contains the pointer that points to the parent frame (that, as we shall see, affects where the output variables are stored.) Then, according to the 'C' calling convention, arguments that are passed to the function (the equivalent of Prolog input variables) are pushed onto the CPU's stack one by one. When the execution of the procedure is complete, the instruction pointer returns to the parent procedure. If the procedure is a branch of deterministic predicate, the stack frame corresponding to the procedure is is removed from the stack. If not, it remains on the stack so that information can be reused if necessary.

Thus, when the argument is an output argument, the function cannot simply write the value into a location on the stack as it does with an input variable, since the stack is cleaned up when the routine finishes, and the vales would lost. Instead, the Prolog interpreter/compiler pushes onto the stack the address of where the value is to be written. This is just what 'C' does, of course, for what it passes is the address, i.e., a pointer, to the variable instead of the variable itself. The address is that of stack location of the variable in the parent predicate. Thus the output of a procedure sets the local variables in the procedures parent procedures.

When the clause or clauses (the Prolog code) for a predicate occur in the same module as the predicate declaration, the Prolog interpreter/compiler can find the flow pattern for any predicate. All it needs to do is to determine whether a variable is bound (has a value assigned to it when the predicate is called) or free (takes on a value a value as the predicate does its work). The interpreter/compiler will assign to bound variables the type 'integer' (or whatever type is appropriate), while to those that are not bound it will assign the type 'pointer to integer' (or whatever type is appropriate). Because the PDC Prolog compiler does extensive flow analysis during compilation (this is one of its strengths), the code it compiles is very efficient, with considerable saving in memory and time. The means that traditional Prolog use to access variables, through pointers, doubles the time and memory requirements for any input variables over PDC Prolog requirements.

Furthermore, because the PDC Prolog compiler does this flow analysis as it is compiling, it can even spot when a predicate is used with different flow patterns--with different parameters bound and free--and compile different code for the different cases. (This means that a predicate sometimes can perform reverse operations. For example, the predicate that we normally use to concatenate strings to break stings down. The predicate call **str_cat("temp", "fil", FileName)** will concatenate the two strings **"temp"** and **"fil"** and bind the variable **FileName** to the result, **"temp.fil"**. However, if the predicate call as follows, **str_cat("temp", Ext, "temp.fil")**, with the first and third values bound and and the second free, the interpreter will decide what value needs to be concatenated with **"temp"** to produce **"temp.fil"**--in other words it will extract **"temp"** from **"temp.fil"**, and bind the variable **Ext** to **".fil"**). The predicate "**power()**" given above is an example of a predicate for which rules are given for different flow patterns.

However, when a the clause or clauses for a predicate appear(s) in one module and the predicate is called in another, the intrepreter/compiler cannot decide, whether the parameters for that predicate are bound or free, since it does not have access to the code in the clauses for that predicate, i.e., the conditions that must be true for the predicate to be true. So we have to tell the interpreter/compiler whether to allocate memory for an integer (or whatever the appropriate type) or whether to allocate the memory for a pointer to an integer (or what the appropriate type). We must include a statement of the flow pattern along with a declaration of the parameter types when we delare the predicate. This is what the letters in parenthesis after the predicate name and parameter list do: the 'i's indicate bound variables, and the 'o's indicate free variables, or input and output variables, variables that already have values when the predicate is called and variables that acquire values as the predicate does its tasks. Seeing a parameter flagged as an input parameter of certain type, the interpreter/compiler will reserve

memory for that type (for a '**char**', if it is parameter is declared to be of type '**char**' and flagged as an input variable) while, if it sees a parameter flagged as an output parameter of the certain type, the interpreter/compiler will reserve memory for a pointer to that type (for a pointer to a '**char**' if a parameter is declared to be of type '**char**' and flagged as an output variable.)

Another feature some readers have noticed is that the Prolog predicate declarations have no return type. This is because Prolog functions have no return value--and if the predicate does bind a variable to a value--assign it value--it returns that value in its accompanying parameter list. (This is no longer entirely true; since the release of version 3.30, PDC Prolog has allowed predicates to have return values. It has provided for return values to make it possible to send and return values to the Windows operating system. Allowing functions to have a return value allows PDC Prolog programmers to copy the function declarations from the 'C' library almost verbatim--you have to substitute "**integer**" for "**int**", and, under appropriate conditions, "**string**" for "**char***" and occasionally a list type for an array type. Nonetheless the provision is not really "in the spirit of Prolog," and I prefer to restrict my use of the predicates with return values to programming for Windows using standard Windows functions).

To conform to Prolog protocol requires rewriting all the 'C' functions that have a return value. For most functions this is a trivial chore. Whenever a 'C' function returns a value, you create a predicate declaration for it by transcribing the name of the 'C' function, making it the name of the Prolog predicate, and specifying, in parentheses following the predication name, the same parameter types (*mutatis mutandis*), in the same order, as appear the 'C' function's parameter list. You then add an additional item at the end of its parameter list of the same type as the return value of the 'C' function. Since the values that 'C' functions return are really pointers, the additional parameter must be of the pointer type.

Take for example the routine from the Gruber Fastgraph library that finds the best available video mode. The Fastgraph manual declares the routine as:

int fg_bestmode(int, int, int);

where:
- the first parameter is the minimum horizontal resolution you want,
- the second is the minimum vertical resolution you want,
- and the third is the number of video pages you want.

If '**fg_bestmode()**' finds a video mode that conforms to the requirements you stipulate, it returns the mode number for the best video mode available (according to the set of criteria on which the function makes its decision.) If it cannot find a video mode on the system that you are using that conforms to your stipulations, it returns -1.

Prolog, by contrast, would return the video mode in an output parameter. We can make 'C' do this by writing the function in the following way:

```
void fg_bestmode_0(int horiz, int vert,int page, int * mode)
    {
    extern int fg_bestmode(int, int, int);

    *mode = fg_bestmode(horiz, vert, page);
    }
```

The funny 'underbar/0' at the end of the predicate name is one of PDC Prolog's conventions. The PDC Prolog compiler treats predicates with the same name but different flow patterns as different predicates. Therefore every flow-patterns for every predicate must be identified differently in the low-level code the compiler builds. To allow for this, the PDC Prolog compiler attaches the suffix '_0' to the predicate name as it builds the low-level code for the first

flow pattern. If the predicate is subsequently used with a second flow pattern, the compile attaches the suffix '_1' to it, and so on. So calling 'C' functions with the suffix already attached to it aligns the calling convention with the compiler's activity. For those who know the origin of the convention, it also serves as reminder to keep flow patterns in mind when working with global predicates.

We can handle the Fastgraph 'C' function that returns the maximum x coordinate in screen space similarly. The Fastgraph manual declares it, for the 'C' library, as:

int fg_getmaxx(void);

The function normally returns a pointer to value; we rewrite it so that its return value appears as a formal parameter (a parameter in the parenthesis after the function name), as a pointer to a value of the same type as the return value of original function. We do this by rewriting it:

```
void fg_getmaxx_0(int * maxx)
    {
    extern int fg_getmaxx(void);

    *maxx = fg_getmaxx();
    }
```

Similarly, the Fastgraph manual declares the Fastgraph function that translates a screen space into world space equivalent as:

```
    double fg_world(int ix);
```

Again we rewrite the function so that instead of having the function return a pointer to a double, it passes the value back through in the parameter list, again through a pointer to a double.

```
void fg_xworld_0(int screen_space_x, double * world_space_x)
        {
    extern double fg_xworld(int);

    *world_space_x = fg_xworld(screen_space_x);
    }
```

Jumping ahead of ourselves a bit, we note that a double is not what a PDC Prolog's standard types. When constructing our 'header' file, we'll have to use '**real**' as our type specification instead of 'integer', to accommodate values greater than 32,267 or 65535.

We now can rewrite 'C" functions that return a pointer to a single value. What about those functions that return an array of values rather than a single value? This is a little thornier, but not very much. It is a little thornier because Prolog doesn't have an array type. We could build the procedures, of course, but it's unnecessary. The author of Prolog, Alain Colmeraurer wrote it for artificial intelligence research--for his research in natural language processing. This provenance has encouraged people to suppose that it is an airy-fairy language suited to a research laboratory but not to the real world. To the contrary, because it originated as an artificial intelligence language, it has marvellous built-in list processing capabilities, and list processing is well suited for graphics programming (and for programming musical applications as well). Just look at how many of the predicates in the algorithmic composer example use the list structure, and you will get an inkling how important it is. Imagine the overhead in using 'C', and having to pass around array lengths to construct functions for accessing the array

elements, etc. It was not for nothing that Autocad choose LISP, the other well-known list processing language, for its language. A list a very useful data-structure for stipulating the vertices of a polygon, or the control points for a Bezier line, or the points on a polyline, or the list of frequencies (notes) the computer should play. In 'C' you use arrays for instead, and you have to know about how large the array should be, and allocate enough memory for it, and if you're looping through all its members, you have to know, or be able calculate it (probably using some statement like '**for (int i=0; i < ((sizeof(array1) / (sizeof(int)); i++)**'), while the Prolog machine takes care of all of this for you.

There is another advantage to Prolog, reminiscent of C++, though we're not using it here, and that is in Prolog you can model the domain that you are programming for using user-defined domains that have some resemblance to objects in C++. It was this capacity that first attracted to me Prolog, before C++ had gained much popularity, and while writing graphics routines for three dimensional transformations of figures and three-dimensional display (including shading effects), I used the provision extensively. You can define domains like:

twoD_point = point2(integer,integer)
threeD_point = point3(integer,integer,integer)
homogen_point = pointH(integer,integer,integer,integer)

Or

twoD_vector = vector2(real,real)
threeD_vector = vector3(real,real,real)
homogen_vector = vectorH(real,real,real,real)

and then declare predicates like

add_vector2(twoD_vector,twoD_vector,twoD_vector

and

line_between(twoD_point,twoD_point)

You can even overload predicates. You can declare a predicate '**line_between**' that takes two twoD_points, and another predicate that takes a list of twoD_points by first declaring a domain

twoD_pointlist = twoD_point*

where '*' means what it does in Backus-Naur Form, i.e., an arbitrary number of twoD_points

then declaring the predicate

line_between(twoD_pointlist)

The PDC Prolog compiler recognizes these as two different predicates since they have different arities--different numbers of parameters, and treats them as separate predicates. The Prolog programmer can think of them in object-oriented terms, as an overloaded "function," i.e., an overloaded predicate. You can even construct user-defined domains using other user-defined domains. For example, you can define a domain (a user-defined data type) '**point**' as:

point = twoD_point; threeD_point; homogen_point

That is, a point is either a point in two-dimensional space, a point in three-dimensional space, or a point in homogeneous coordinates, then write a clause '**join(point,point)**' that, given the proper checking, will respond differently if given a pair of twoD_points, a pair of threeD_points and a pair of homogen_points, or even, points of different dimensionality (perhaps then the programme issues an error message). This doesn't provide everything the C++ objects provide, but it goes quite a way, and it is convenient and simplifies modeling problems, and best of all, it doesn't have the enormous overhead that C++'s (admittedly more powerful) objects have. For example it enables us to call a predicate that might, for example, convert between two domains, perhaps

threeD_to_twoD(threeD_point, twoD_point),

and let the statements that make up the clause handle the nitty-gritty stuff. The resemblence to object-oriented methods should be clear. Nor should the resemblence come as a surprise, since object-oriented programming descends partly from the familiar entity-attribute-relation (E-A-R) diagram so commonly used in relational database programming, and one of Prolog's fortés as a language is creating relational databases. While these capabilities are important, I have not used these very powerful capabilities in any of the modules I have included, in order to make the Prolog predicates resemble the 'C' functions as much as possible.

To handle the data that some Fastgraph 'C' functions return as arrays, we have to convert the array to a list; and because lists are vastly more convenient than arrays, we want to be able to handle the data a list in our Prolog programme, and have the Prolog-'C' interface handle the work of converting the list to an array that the 'C' library function we call can handle. Fortunately, the folks at Prolog Development Center have already done the dirty-work for us, and we can call on their code to do it for us. Here for example is how we handle the Fastgraph function '**fg_polygon()**'. This polygon function is declared in 'C' as follows:

void fg_polygon(int * ix_array, int * iy_array, int n);
where:

*ix_array is an arbitrary-length array of screen space x-coordinates,

*iy_array is an arbitrary-length array of screen space y-coordinates,
and n is the number of vertices in the polygon, (a nasty and pesky animal that Prolog doesn't need to pass a around with lists).

The function "**fg_polygon()**" must be able to convert homogenous aggregates of integers back and forth between lists and arrays. Functions that convert a list of integers into an array of integers and an array of integers into a list of integers are:

```
/*-------------------------------------------------------------
for converting between lists and arrays of integers
This part is taken from PDC Prolog Reference Guide
Release 3.2x page 98
Release 3.3x pages 103-104
-------------------------------------------------------------*/

#define listfno  1
#define nilfno   2

void *MEM_AllocGStack(unsigned);
```

```
typedef struct ilist {
   char Functor;
   int Value;
   struct ilist *Next;
} IntList;


int IntListToArray(IntList *List,int **ResultArray)
{   /* Convert a list to an array placed on the global stack */
   IntList *SaveList = List;
   int *Array;
   int  i = 0;

 /* Count the number of elements in the list */
   for(i=0; List->Functor==listfno; List=List->Next)
      i++;

   Array = MEM_AllocGStack(i*sizeof(int));
   /* Allocate the needed memory */
   List = SaveList;
   /* Transfer the elements from the list to the array */

   for(i=0; List->Functor==listfno; List=List->Next)
      Array[i++]=List->Value;

   *ResultArray=Array;
   return(i);
}

void IntArrayToList(int Array[],int n,IntList **List)
 /* Convert an array to a list */

{
   int i;

    for (i=0; i<n; i++)  /* Allocate a record for each element */
          {
           IntList *p=*List=MEM_AllocGStack(sizeof(IntList));
               p->Functor=listfno;
               p->Value=Array[i];
           List=&(*List)->Next;
     }

   {  /* Allocate the last record in the list */
      IntList *p=*List=MEM_AllocGStack(sizeof(char));
      p->Functor=nilfno;
   }
}

void incr_intlist_0(IntList *InList,IntList **OutList)
{  /* Increment all values in a list */
```

```
  int i, n, *Array;
  n=IntListToArray(InList,&Array);
  for(i=0; i<n; i++) Array[i]++;
  IntArrayToList(Array,n,OutList);
}

/*-------------------------------------------------------------
      end of the Prolog Development Center's code
-------------------------------------------------------------*/


void fg_polygon_0(IntList *xlist, IntList *ylist)
{
    int *xarray, *yarray, n=0, m=1;
    extern void fg_polygon(int *, int *, int);

    n = IntListtoArray(xlist,&xarray);
    m = IntListToArray(ylist,&yarray);

    if (m != n)
    {
        printf("\nUnequal number of x and y coordinates in");        printf("\nfor polygon.");
        exit(0);
    }
    else
    fg_polygon(xarray,yarray,n);
}
```

One need not include the check that list lengths are equal. I have included it because, when you do not have to send list lengths to the predicates, it would be easy to omit an x- or y-coordinate, and there is otherwise nothing in the procedure to help you spot the problem. If one were so inclined (and it would certainly show up the beauty of Prolog) he could rewrite the predicate so that it is takes a list of twoD_points and you could call the predicate as '**fg_mypolygon(VertixList)**'. This is what I did when three-dimensional routines I mentioned above, and I think that this really is the way it should be done. Furthermore, it is dead-easy to write the code to travel down the point list extracting all the x-coordiante values and y-coordiante values and building lists for each, then handing the two integer lists to '**fg_polygon_0**', which translates them into 'C''s awkward array structure, and plots them. My preference has been to align the Prolog predicates closely with their 'C' equivalents in the Fastgraph, so that you don't need to keep scribbling little notes into Gruber's excellent manuals. With the many the 'C' examples it contains, it serves fine as Prolog-Fastgraph reference manual, so long as you you are scrupulous about transforming you 'C' functions into Prolog predicates by applying a set of rules consistently. The rules I have stipulated above are that you transfer the 'C' return values to the predicate's parameter list and use lists instead of arrays for storing homogeneous aggregates of coordinate values. Using lists mercifully makes it unnecessary to keep track of the size of the aggregate, and that alone gives Prolog a considerable advantage over the 'C', one that, in my view, is sufficient grounds for considering migrating to Prolog.

The code above, written by the Prolog Development Center (that authors of PDC Prolog) includes a call to functions named "**MEM_AllocGStack**" and "**MEM_ReleaseGStack**". These are memory-handling predicates that are included in PDC Prolog's 'prolog.lib.' When Prolog

calls "**fg_polygon()**", it does not allocate space for the for the array itself. The task of allocating the memory space for the array falls on the 'C' function itself. However, if 'C' and Prolog were attempt to allocate memory space simultaneously, the result would surely be internal dickering overing memory and system crashes. So the memory allocation is done through "Prolog" routines (which are really just 'C' functions) that are installed in 'prolog.lib' and over which the Prolog compiler has control. Prolog keeps all "complex variables" such as strings and "compound objects" (user-defined data types, often aggregates of other data-types rather like the data types that 'C' constructs using the "**typedef struct { }**" series. The "**twoD_point**" and "**threeD_point**" data types defined above are compound objects) on a global stack. A list and an array are compound objects and so must be put on the global stack (or gstack). The functions for handling allocation of Prolog's global stack are:

void *MEM_AllocGStack(unsized num_bytes_to_be_allocated)
void *MEM_ReleaseGStack(void *p)
void *MEM_MarkGStack(void)

"**MEM_AllocGStack()**" allocates the number of bytes specified in num_bytes_to_be_allocated on the top of Prolog's global stack/ This space is automatically released whenever the programme backtracks to a point before the place where this memory was allocated.
"**MEM_ReleaseGStack()**" sets top of the stack to the address specified in '**p**'.
"**MEM_MarkGStack()**" returns a pointer to the top Prolog's global stack that makes possible manual recovery of GStack space. PDC provides comparable procedures for handling the heap.

The procedure for converting an array of doubles into a list of reals is no more difficult. We have to do this in cases like the Fastgraph function '**fg_polygonw()**' that is declared as:

void fg_polygonw(double *x_array, double *y_array, int x);
where:
*x_array is an arbitrary-length array containing the world space x-coordinates of the polygon vertices,
*y_array is an arbitrary-length array containing the world space y-coordinates of the polygon vertices,
and n is another of those nasty necessities of 'C', the number of vertices.

The predicate must be able to convert a list of **real**s into an any array, or an array of reals into a list. Code for doing so follows:

```
/*-------------------------------------------------------------for converting between lists and arrays of
reals
Patterned on code found the PDC Prolog Reference Guide,
release 3.2, pg 98, release 3.3, pp 103-1-4
-------------------------------------------------------------*/

#define listfno  1
#define nilfno   2

void * MEM_AllocGStack(unsigned) ;

typedef  struct rlist {
  char lFunctor;
  int Value;
```

```
  struct ilist *Next;
} RealList;

int RealListToArray(RealList *List, double **ResultArray)
{   /* Convert a list to an array placed on the global stack */
  RealList *SaveList = List;
  double *Array;
  int  i = 0;

 /* Count the number of elements in the list */
  for(i=0; List->Functor==listfno; List=List->Next)
    i++;

  Array = MEM_AllocGStack(i*sizeof(double));
  /* Allocate the needed memory */
  List = SaveList;

  /* Transfer the elements from the list to the array */

  for(i=0; List->Functor==listfno; List=List->Next)
    Array[i++]=List->Value;

  *ResultArray=Array;
  return(i);
}

void RealArrayToList(double Array[],int n,RealList **List)
 /* Convert an array to a list */
{
  int i;

  for (i=0; i<n; i++)  /* Allocate a record for each element */
   {
    RealList *p=*List=MEM_AllocGStack(sizeof(RealList));
    p->Functor=listfno;
    p->Value=Array[i];
    List=&(*List)->Next;
  }

  {  /* Allocate the last record in the list */
    RealList *p=*List=MEM_AllocGStack(sizeof(char));
    p->Functor=nilfno;
  }
}


/*      end of code patterned after PDC Prolog's
-------------------------------------------------------------*/
void fg_polygonw_0(RealList *xlist, RealList *ylist)
{
    double *xarray, *yarray, n = 0, m = 1;
```

```
    extern void fg_polygonw(double *, double *, int);


    n = RealListtoArray(xlist,&xarray);
    m = RealListToArray(ylist,&yarray);

    if (m != n)
    {
        printf("\nUnequal number of x and y coordinates in");
        printf("\nfor polygon.");
        exit(0);
    }
    else
    fg_polygonw(xarray,yarray,n);
}
```

The final wrinkle that we face is that some arrays should not be converted into lists, but into binary blocks. It's relatively simple to get a binary block from 'C' and treat it as a binary block in Prolog, as PDC Prolog now includes a domain "**binary**" that handles byte arrays of specified length. Earlier versions of PDC Prolog included "**readblock**/2" and "**writeblock**/2" predicates, for reading or writing byte-sequences that might include zeros or end-of-file markers from or to a file. In earlier versions of PDC Prolog, however, the binary block masqueraded as a string. With the effort to make PDC Prolog a tool for programming Microsoft Windows, binary blocks took on such importance that they became a domain in their one right. A binary block no longer has to pass itself off as a string. The availability of binary blocks enables us to pick up a byte array from a 'C' function and to access its components with the predicate, also introduced with version 3.3, '**getbyteentry()**/2'. This is how I have handled '**fg_getimage()**' and '**fg_getmap()**'. However, passing a byte array from Prolog to 'C' is a little harrier. Let us take the case of the Fastgraph function fg_setDACs(), that is declared as:

void fg_setdacs(int start, int count, char* values)
where :
'start' is the starting video DAC register, a value between 0 and 255,
'count' is the number of contiguous DAC registers to define, between 1 and 256 (Gruber provides for wrapping if the value exceeds 256),
and 'values' is a char (unsigned byte) array containing colour components. The first three bytes of this array contain the red, blue and green components for DAC register 'start,' the second three bytes contain the red, blue and green components for DAC register 'start+1,' etc.

If we were willing to relax our standards a little and tolerate handling predicates that pass byte arrays differently than other predicates, we could write and use a special Prolog predicate. Let's call it '**setdacs_prolog()**.' The predicate '**setdacs_prolog()**' would contain the code for converting the integer values that represent the red, blue, green DAC values into binary values, then sending them off the result to a binary array, and the calling code for the Prolog version of the Fastgraph library function '**fg_setdacs()**', A call to setdacs_prolog()" would then handle the job of converting integer colour values into binary value, storing the new binary values in a binary array, and passing the binary array to the Prolog version of '**fg_setdacs()**', thus setting a new palette. However the use of the predicate would impose on the user the need to remember to call this library function in a special way. I prefer to have total compatibility between the Prolog "version" of the Fastgraph library and the 'C' version that Ted Gruber software provides,

so that when we consult the Fastgraph manual on how to use a function, we learn everything that we need to know in order to use it, as there are no undocumented tasks (such as remembering that '**fg_setdacs()**' should not be called directly from Prolog, but through the predicate '**setdacs_prolog()**'). This practice also ensures that the examples in Gruber's excellent Fastgraph User's Guide work pretty much as we find them. I have not provided the code for a '**setdacs_prolog()**' predicate, as I remain adamant about maintaining close compatibility with Gruber's Fastgraph library. However a '**setdacs_prolog()**' predicate obviously has real advantages, and some may consider that the advantages outweigh the disadvantages.

The simplest and most elegant way to specify the DAC values would be in a list. (I would prefer would be to use a Prolog compound domain to model the structure of a DAC register, by defining a new Prolog domain--i.e., data-type:

dac_register = dac(integer,integer,integer)

However, I've ruled out using this feature of Prolog, one of its most elegant, again to conform closely to the Fastgraph 'C' function declarations.) Given that reluctance, the best thing to do would use an integer list to represent what in 'C' will become and array of chars, and convert the integerlist into an array of **bytes** in the 'C' function. This keeps up the principle--on which we must maintain some flexibility, that Prolog uses lists where 'C' uses arrays and avoiding 'C''s extraordinary fastidiousness about efficiency. We behave wantonly and use the most natural-seeming data-type to represent a datum. If we want to represent red, green or blue value that can range between 0 and 63 we can, because of the restricted range of the values, use a char to represent it, and whenever we use an integer to represent the value cast the integer value as a char. Nonetheless, the most natural data-type to use to represent a whole number value, whether it between 0 and 63 or between 0 and 32000, is an integer. Who wants to be bothered remembered that there are only 64 for values that numerical parameter can have, and hence that we can save time and memory by allotting this numerical value the bit-width of a character instead of that of an integer? If you care to be so fastidious, you are welcome to 'C.' I think the time and memory saving is not worth pain, particularly if the programme you are writing will only run a few times. If we apply maintain consistency on this, we can assume, when we read the specification for the 'C" function in the Fastgraph manual, and see that the 'C'-function uses the "**char**" as a numerical value, that we used the "**integer**" data-type to represent what 'C' used the "**char**" data-type to represent. So we make a rule of this, just as we make a rule that we represent the return value of 'C' functions in the parameter list of the Prolog calling function, or that use to strings to represent pointers to chars when appropriate.

How do we handle the conversions?

```
/* setdac_p.c */

/*-----------------------------------------------------------------for converting back and forth between lists
of integers and arrays of bytes and This part is based upon code written by Prolog
Development Center and published in PDC Prolog Reference Guide (Release 3.2x page
98
Release 3.3x pages 103-104)
 ----------------------------------------------------------------*/
#define listfno  1
#define nilfno   2

void *MEM_AllocGStack(unsigned);
```

```
typedef struct ilist {
  char Functor;
  int Value;
  struct ilist *Next;
} IntList;


int IntListToByteArray(IntList *List,int **ResultArray)
{   /* Convert a list to an array placed on the global stack */    IntList *SaveList = List;
  int *Array;
  int  i = 0;

 /* Count the number of elements in the list */
  for(i=0; List->Functor==listfno; List=List->Next)
    i++;

   Array = MEM_AllocGStack(i*sizeof(int));

/* Allocate the needed memory */
        List = SaveList;
/* Transfer the elements from the list to the array */

  for(i=0; List->Functor==listfno; List=List->Next)
    Array[i++]= (char) List->Value;

  *ResultArray=Array;
  return(i);
}

void ByteArrayToIntList(int Array[],int n,IntList **List)
 /* Convert an array to a list */
{
  int i;

  for (i=0; i<n; i++)  /* Allocate a record for each element */    {
    IntList *p=*List=MEM_AllocGStack(sizeof(IntList));
    p->Functor=listfno;
    p->Value=(int)Array[i];
    List=&(*List)->Next;
  }

  {  /* Allocate the last record in the list */
    IntList *p=*List=MEM_AllocGStack(sizeof(char));
    p->Functor=nilfno;
  }
}

void fg_setdacs_0(int start, int count, IntList *valslist)
   {
   char *dacs_array;
```

```
    extern void fg_setdacs(int, int, char *);


    IntListToByteArray(valslist,&dacs_array);
    fg_setdacs(start, count, dacs_array);
    }
```

We handle data retrieved from the dac registers similarly. We fetch the data in the form of a byte array, then convert the contents of the array into an integer list, which becomes the output parameter of the Prolog version of "fg_getdacs()"

```
void fg_getdacs_0(int start, int count, IntList ** values)
    {
    char * dacregisters;
    extern void fg_getdacs(int, int, char *);

    fg_getdacs(start,count,dacregisters);
    ByteArrayToIntList(dacregisters,count,values);
    }
```

All the functions in the Fastgraph library that have return values must be rewritten on this model. All the code for doing so appears at the end of the article. The code appears in many tiny modules that we compile to separate '.obj' modules that we put into the library one by one, so that code for the whole lot of them does not have to be linked into the '.exe' file whenever any any one of them is called. A few rules have to followed when compiling the modules. PDC Prolog uses the large memory model, and so the C modules have to be compiled using the large memory model. It is prudent to align data on items on byte boundaries, as this is necessary of any compound data structures (like **point(integer,integer)**) are used. You never know when you might get the urge to use this capability, as it adds tremendously to your power to model features of the real world, so you might as well prepare for it. It is also prudent to generate a standard stack frame, since a stack frame is required if you call any non-deterministic predicates. The stack overflow check for 'C' must be switched off. The default type for characters must be unsigned.

Once the 'C' and 'obj' code for these modules is available, we have to compose a 'header' file so that all Prolog compiler will know not to quit if it finds a predicate with no clauses for it in the module it is compiling--so that it will know that the linker can expect to find the code for those predicates when we link in the Fastgraph library. The 'header' file, with its conventional '.pre' extension, is given at the end of the article. It has the name 'fastgraf.pre.' There are two things to note concerning the file. The first is that after the flow diagram, we put the name of 'C' function that the Prolog predicate calls with the state "as [C-Name]", where [C-Name] is the 'C' function name. The 'C' name is preceded by an underbar. This is because 'C' compilers, for some peculiar reason, generate names that are preceded by an underbar, and we are simply aligning our practice with this arcane 'C'practice. We must also not forget that in the cases of functions that we have rewritten, we are calling the rewritten version (which, by my convention will have an 'underbar-0' as a suffix for the name), and not Mr. Gruber's version directly. Finally, we inform the compiler that the function the Prolog predicate calls will pass its parameters according to the 'C' calling convention. We do this by inserting the statement "language c" before the 'C' name. (Thus the "language 'C' statements" thus serve much the same role that the reserved words "**PASCAL**"and "**_cdecl**" do in C'.)

A response file that can used to collect the PDC Prolog-Fastgraph interface together in

the new Prolog Fastgraph library is provided below. Ted Gruber Software provides libraries for several different 'C' compilers and memory models. PDC Prolog requires that the large memory model be used, and so the Fastgraph library for the large memory model is the proper choice. If you are using the Fastgraph/Light shareware version of the Fastgraph library, the library is named 'fgll.lib'. If you register, and so eliminate the bother screen that comes up whenever you initialize a graphics mode using Fastgraph/Light, you get two libraries, one that duplicates the functions in the Fastgraph/Light library and the other a library that enables the to specify a world coordinate system instead of using the hard ware-defined screen (device) coordinate system. These two libraries are named 'fgl.lib' and, if you are using a Borland C compiler, 'fgtcl.lib'; if you are using a Microsoft C compiler, 'fgmscl.lib'; and, if you are using a Power C compiler 'fgpcl.lib.' To create a new library that includes the Prolog-Fastgraph interface from the Fastgraph library that Ted Gruber Software provides, assuming that you are using a Borland 'C' or 'C++' compiler and that 'tlib.exe,' the Fastgraph library, and 'fastgraf.rps' are in the current directory, you can, once you have saved a copy of 'fgl.lib' use the following command:

tlib fgll.lib @fastgraf.rps, profast.lst

to bundle all our '.obj' files into the library. You should then rename the library (I've named mine 'profast.lib') to distinguish the fastgraph library that included the Prolog interface from the 'fgll.lib' that Ted Gruber Software provides. If you are using a registered version of Fastgraph, you must first install the interface between Prolog and 'fgl.lib' in 'fgl.lib.' Under the same conditions set out above, and using the "fastgraf.rps" response file, once you have saved a mint copy of 'fgl.lib', you issue the command

tlib fgl.lib @fastgraf.rps, profast.lst

and rename the new 'fgl.lib' that 'tlib.exe' produces as 'profast.lib' or something else that distinguish it from the 'fgl.lib' file that Gruber provides. Next, you create a new extended library that contains the interface between Prolog and 'fgtcl.lib.' A response file is provided for this purpose as well. Presuming that 'tlib.exe' and 'fgtcl.lib' and 'xfast.rps' are in the current directory, that your '.obj' files are in "c:\fastgraf\store" and that you have saved a copy of 'fgtcl.lib', issue the following command:

tlib fgtcl @xfast.rps, xprofast.lst,

and then rename the new library that 'tlib.exe' produces 'xprofast.lib,' or some other unique name that distinguishes it from 'fgtcl.lib,' the extended capability file that Ted Gruber Software provides.

Now you're ready to call the Fastgraph library functions from PDC Prolog. All you need to is to "**include**" the 'fastgraf.pre' module in your programme, then write your programme, calling the Fastgraph predicates (or any predicates that you installed in a library and declared in '.pre' file that have included in the project) as though the predicates are part of the basic Prolog language.

An couple of examples follow. The first provides a very fancy way of setting the selecting the video mode, using a convenient menu that eliminates the necessity of remember all those video mode numbers. It is also quite robust for, if the user chooses a mode that is not available, the programme lets him or her choose another. The second example displays a '.pcx' file.

/* initfast.pro */

```
include "c:\\fastgraf\\profast\\fastgraf.pre"
include "c:\\prolog\\ui\\menu.pro"
  % the "menu.pro" file is part of the Prolog Development Centre
  %     Prolog Toolkit--if you do not have the Toolkit, see note
  % below.

/*-------------------------------------------------------
select_mode(RecommendedMode)--let the user choose a supported video mode.  Note:
the responses to requests for the various modes accord with my system.  Individual
users will have to make alterations for their systems.  Since Fastgraph does not provide
suppport for SuperVGA modes, the code below will work for most systems with a VGA
board.
-------------------------------------------------------*/

PREDICATES
select_mode(integer)
check_mode(integer)
select_mode_aux(integer,integer)
assert_graphics_mode(integer)

CLAUSES

select_mode(RecommendedMode):-
    check_mode(3),
    makewindow(1,31,31,"[What video mode do you want?]",
               5,10,14,50),
    writef("\n Recommended mode is Mode %.", RecommendedMode),
    menu(8,18,31,31,[
         " ‖ 320x200x16  ‖   0D   ‖  13    ‖ ",
         " ‖ 320x200x256 ‖   13   ‖  20    ‖ ",
         " ‖ 320x400x256 ‖   13   ‖  21    ‖ ",
         " ‖ 640x200x2   ‖   06   ‖  06    ‖ ",
         " ‖ 640x200x16  ‖   0E   ‖  14    ‖ ",
         " ‖ 640x350x16  ‖   10   ‖  16    ‖ ",
         " ‖ 640x480x16  ‖   12   ‖  18    ‖ ",
         " ‖ 720x348x2   ‖   11   ‖  11    ‖ "],
         " resolution   hex    decimal",9,Choice),
         removeallwd(),
         assert_graphics_mode(Choice),
         select_mode_aux(RecommendedMode,Choice).

/* Note: the predicate "menu()" is part of Prolog Development
 * Center's  Prolog Toolkit.  If you do not have the toolkit, you
 * will have sustitute:
*
* write("\n For 320x200x16  Mode 13 -- Enter 1 "),
* write("\n For 320x200x256 Mode 20 -- Enter 2 "),
* write("\n For 320x400x256 Mode 21 -- Enter 3 "),
```

```
* write("\n For 640x200x2   Mode 06 -- Enter 4 "),
        etc.  for remaining video modes
* readint(Choice),
*/

assert_graphics_mode(1):-!,
    retractall(graphics_mode(_),fastgraph),
    assert(graphics_mode(13),fastgraph).

assert_graphics_mode(2):-!,
    retractall(graphics_mode(_),fastgraph),
    assert(graphics_mode(20),fastgraph).

assert_graphics_mode(3):-!,
    retractall(graphics_mode(_),fastgraph),
    assert(graphics_mode(21),fastgraph).

assert_graphics_mode(4):-!,
    retractall(graphics_mode(_),fastgraph),
    assert(graphics_mode(6),fastgraph).

assert_graphics_mode(5):-!,
    retractall(graphics_mode(_),fastgraph),
    assert(graphics_mode(14),fastgraph).

assert_graphics_mode(6):-!,
    retractall(graphics_mode(_),fastgraph),
    assert(graphics_mode(16),fastgraph).

assert_graphics_mode(7):-!,
    retractall(graphics_mode(_),fastgraph),
    assert(graphics_mode(18),fastgraph).

assert_graphics_mode(8):-!,
    retractall(graphics_mode(_),fastgraph),
    assert(graphics_mode(11),fastgraph).

select_mode_aux(_,1):-
    mode13_avail(true),!,
    fg_setmode(13).

select_mode_aux(RecommendedMode,1):-!,
    makewindow(2,31,31,"[Graphics Error]",5,10,10,60),
    write("\n\t Graphics Mode 13 not available on you system"),
    nl,
    sleep(10),
    write("\n\n\t Press any key to make another selection"),
    readchar(_),
    removewindow(2,1),
    select_mode(RecommendedMode).
```

```
select_mode_aux(_,2):-
    mode20_avail(true),!,
    fg_setmode(20).

select_mode_aux(RecommendedMode,2):-!,
    makewindow(2,31,31,"[Graphics Error]",5,10,10,60),
    write("\n\t Graphics Mode 20 not available on you system"),
    nl,
    sleep(10),
    write("\n\n\t Press any key to make another selection"),
         readchar(_),
    removewindow(2,1),
    select_mode(RecommendedMode).

select_mode_aux(_,3):-
    mode21_avail(true),!,
    fg_setmode(21).

select_mode_aux(RecommendedMode,3):-!,
    makewindow(2,31,31,"[Graphics Error]",5,10,10,60),
    write("\n\t Graphics Mode 21 not available on you system"),
    nl,
    sleep(10),
    write("\n\n\t Press any key to make another selection"),
    readchar(_),
    removewindow(2,1),
    select_mode(RecommendedMode).

select_mode_aux(_,4):-
    mode06_avail(true),!,
    fg_setmode(6).

select_mode_aux(RecommendedMode,4):-!,
    makewindow(2,31,31,"[Graphics Error]",5,10,10,60),
    write("\n\t Graphics Mode 6 not available on you system"),
    nl,
    sleep(10),
    write("\n\n\t Press any key to make another selection"),
    readchar(_),
    removewindow(2,1),
    select_mode(RecommendedMode).

select_mode_aux(_,5):-
     mode14_avail(true),!,
     fg_setmode(14).

select_mode_aux(RecommendedMode,5):-!,
    makewindow(2,31,31,"[Graphics Error]",5,10,10,60),
    write("\n\t Graphics Mode 14 not available on you system"),
    nl,
    sleep(10),
```

```
    write("\n\n\t Press any key to make another selection"),
    readchar(_),
    removewindow(2,1),
    select_mode(RecommendedMode).

select_mode_aux(_,6):-
    mode16_avail(true),!,
    fg_setmode(16).

select_mode_aux(RecommendedMode,6):-!,
    makewindow(2,31,31,"[Graphics Error]",5,10,10,60),
    write("\n\t Graphics Mode 16 not available on you system"),
    nl,
    sleep(10),
    write("\n\n\t Press any key to make another selection"),
        readchar(_),
    removewindow(2,1),
    select_mode(RecommendedMode).

/*  Replace when debugged
select_mode_aux(_,7):-
    mode18_avail(true),!,
    fg_setmode(18).
*/

select_mode_aux(_,7):- fg_setmode(18),!.

select_mode_aux(RecommendedMode,7):-!,
    makewindow(2,31,31,"[Graphics Error]",5,10,10,60),
    write("\n\t Graphics Mode 18 not available on you system"),
    nl,
    sleep(10),
    write("\n\n\t Press any key to make another selection"),
        readchar(_),
    removewindow(2,1),
    select_mode(RecommendedMode).

select_mode_aux(_,8):-
    mode11_avail(true),!,
    fg_setmode(11).

select_mode_aux(RecommendedMode,8):-!,
    makewindow(2,31,31,"[Graphics Error]",5,10,10,60),
    write("\n\t Graphics Mode 11 not available on you system"),
    nl,
    sleep(10),
    write("\n\n\t Press any key to make another selection"),
        readchar(_),
    removewindow(2,1),
    select_mode(RecommendedMode).
```

```
check_mode(Mode1):-
    fg_getmode(Mode2),
    Mode2 = Mode1,!.

check_mode(Mode):-
    fg_setmode(Mode).

/*-----------------------------------------------------------
    initialize -- initialize the Fastgraph environment
-----------------------------------------------------------*/

PREDICATES

fastgraph_init()
check_switch_to_graphics(integer,integer,integer,integer)
recommend_mode(integer,integer,integer,integer,integer,integer)
allocate_pages(integer,integer)

CLAUSES

fastgraph_init():-
    fg_testmode(6, pages,Mode06Avail),
    fg_testmode(11,pages,Mode11Avail),
    fg_testmode(13,pages,Mode13Avail),
    fg_testmode(14,pages,Mode14Avail),
    fg_testmode(16,pages,Mode16Avail),
    fg_testmode(18,pages,Mode18Avail),
    fg_testmode(20,pages,Mode20Avail),
    fg_testmode(21,pages,Mode21Avail),
    retractall(mode06_avail(_),fastgraph),
    assert(mode06_avail(Mode06Avail),fastgraph),
    retractall(mode11_avail(_),fastgraph),
    assert(mode11_avail(Mode11Avail),fastgraph),
    retractall(mode13_avail(_),fastgraph),
    assert(mode13_avail(Mode13Avail),fastgraph),
    retractall(mode14_avail(_),fastgraph),
    assert(mode14_avail(Mode14Avail),fastgraph),
    retractall(mode16_avail(_),fastgraph),
    assert(mode16_avail(Mode16Avail),fastgraph),
    retractall(mode18_avail(_),fastgraph),
    assert(mode18_avail(Mode18Avail),fastgraph),
    retractall(mode20_avail(_),fastgraph),
    assert(mode20_avail(Mode20Avail),fastgraph),
    retractall(mode21_avail(_),fastgraph),
    assert(mode21_avail(Mode21Avail),fastgraph),
    % save("Modes.dba"),

check_switch_to_graphics(Mode06Avail,Mode11Avail,Mode14Avail,
                                        Mode16Avail),
/* Save the current video mode */
    fg_getmode(OldMode),
```

```
    retractall(old_mode(_),fastgraph),
    assert(old_mode(OldMode),fastgraph),

/* Propose the default video mode */
   recommend_mode(Mode18Avail,Mode16Avail,Mode20Avail,
         Mode16Avail,Mode11Avail,RecommendedMode),
/*  Let user choose the video mode he or she wants */
    select_mode(RecommendedMode),

/* Allocate virtual pages as needed */
    fg_initxms(_),
    fg_initems(_),
    init_mouse.
/* end clause */

check_switch_to_graphics(0,0,0,0):-!,
    fg_abort_program(0).

check_switch_to_graphics(_,_,_,_):-!.

recommend_mode(true,_,_,_,_,18):-!.
recommend_mode(_,true,_,_,_,16):-!.
recommend_mode(_,_,true,_,_,20):-!.
recommend_mode(_,_,_,true,_,6):-!.
recommend_mode(_,_,_,_,true,11):-!.


allocate_pages(LoopCount,LoopLimit):-
    LoopCount > LoopLimit, !.

allocate_pages(LoopCount,LoopLimit):-
    fg_allocate(page,Stat),
    Stat = 1,!,
    NewLoopCount = LoopCount + 1,
    allocate_pages(NewLoopCount,LoopLimit).

/*-----------------------------------------------------------
init_hercules -- initialize the Fastgraph environment
-----------------------------------------------------------*/


PREDICATES

init_hercules()
check_switch_to_herc(integer)

CLAUSES

init_hercules():-
    fg_testmode(11,pages,Mode11Avail),
    check_switch_to_herc(Mode11Avail),
```

```
/* Save the current video mode */
    fg_getmode(OldMode),
    retractall(old_mode(_),fastgraph),
    assert(old_mode(OldMode),fastgraph),
    fg_setmode(11),
    retractall(graphics_mode(_),fastgraph),
    assert(graphics_mode(11),fastgraph),
    init_mouse.
/* end clause */

check_switch_to_herc(0):-!,
    fg_abort_program(0).

check_switch_to_herc(_):-!.


/* disppcx.pro */

include "c:\\fastgraf\\profast\\fastgraf.pre"

/*-------------------------------------------------------------
                        dissplay_pcx(FileName)
Becomes true when a '.pcx' image with the name FileName
has been displayed
-------------------------------------------------------------*/

PREDICATES

get_pcx_filename(string)
display_pcx()
display_pcx_aux(string)

CLAUSES

get_pcx_filename(Filename):-
       makewindow(3,31,31,"[Name of PCX File]",7,10,8,60),
       write
       ("\n Enter name of the PCX file you want to display."),
%      readfilename(13,15,31,31,pcx,"",FileName,existing_file),
% This predicate is supplied with with PDC Prolog toolkit -- if
% you have it, uncomment the line about, and comment out the % following;  if you do
not have, leave this predicate the way
% it is.
            readln(FileName),
       removewindow(3,1).

display_pcx_aux(PCX_FileName):-
    fg_pcxsize(PCX_FileName,Width,Length,_),
    fg_getmaxx(MaxX),
    fg_getmaxy(MaxY),
```

```
    HorizontalMargin = (MaxX - Width) / 2,
    VerticalMargin   = (MaxY - Length) / 2,
    fg_setcolor(1),
    fg_move(HorizontalMargin,VerticalMargin),
    fg_allocate(1,_),
    fg_disppcx(PCX_FileName,0,_),
    fg_waitkey().

display_pcx():-
     get_pcx_filename(PCX_FileName),
%     write("\n Enter pictures filemame: ",),
%     readln(PCX_FileName),
     fg_pcxsize(PCX_FileName,Width,Length,_),
     makewindow(2,31,31,"",10,10,8,65),
     writef("\n\ %s\'s size is %  x  %. ",
            PCX_FileName,Width,Length),
     writef("\n\n\t Press any key to continue..."),
     readchar(_),
     removewindow(2,1),
     fastgraph_init,
%     fg_allocate(1,_),
     fg_setpage(0),
     display_pcx_aux(PCX_FileName),
     fg_erase(),
     fg_freepage(1,_),
     old_mode(OldMode),
     fg_setmode(OldMode),
     fg_reset.
```

This example gives us an opportunity to comment on three other feature of Prolog. Most of the entries in the parameter lists of the various predicates listed above have capital letters, though a few have small letters. While Prolog is not a case-sensitive language (Thank Goodness, for that!!!!)--and so you surrender the pleasure that all 'C'-coders know of spending hours looking through pages and pages of code to find if you spelled the sting "Inline" or "InLine" or "INLINE" or, perhaps by mistake "INLine"--the difference between a capital and small letter as the first letter of a parameter is significant. Parameters that begin with capital letter are variables (they may take on different values), parameters that begin with a small letter are constants (that must not take on different values). Constants can of two types--either system-defined constants or user-defined constants. The values for various user-defined constants usuallly appear in the "**constants**" section of the programme, though sometimes they are included in a programme with and "**include**" statement, as the BGI constants are included by calling in the "bgi.pre" file. Readers will note that PDC Prolog treats the filename of an include file just like any other string, so you need to use double back-slashes ("\\"'s) in your paths. The use of strings for filenames is consistent in PDC Prolog, so you needn't remember that when you using "**initgraph()**", for example, you write the "bgi directory path" as a string with double back-slashes, but when you include it, you do not, and use only single back-slashes.

Interspersed in the code above (see the predicates **select_mode_aux()** are dashes in the parameter list. The dash stands for an anonymous variable--a variable that we can have any value whatsoever and the predicate still be true. For example, the **select_mode()** predicate allows the user to choose a a video form a menu of video modes. It takes an input variable,

"RecommendedMode." "RecommendedMode" is the result of testing to see whether the system supports a number of video modes, then recommending the generally preferable mode from among those modes the systems is found to support. It nice thing to be able to tell the user what video mode the system/programme considers the highest mode available when he or she is selecting the mode to use.

When the user chooses a video mode, then "**select_mode_aux()**" is called. One of two things will happen when this predicate is called. Either the system will be able to set the video mode the user chooses, or it will not. (In this programme, we don't do anything so fancy as looking to see which video modes tha machines that system supports and graying the menu entries for those that it does not). If the predicate suceeds in setting the video mode the user has choosen, all is well. If it cannot set that video mode, then the user is given another chance to select a video mode by returning to the beginning of **select_mode()** predicate.

If the programme does need to return to the **"select_mode()**" predicate, it must be handed to the rule through the "RecommendedMode" variable. In order for "**select_mode_aux()**" to be able to hand "**select_mode()**" the variable, "Recommended Mode", it must carry the variable along in its parameter list. However, if "select_mode_aux()" suceeds in setting the video mode, the parameter is not needed, for in that case its job was completed when it enabled "**select_mode()**" to state which video mode is the highest the system supports. Hence, in those clauses in which the predicate "**select_mode_aux()**" suceeds in setting the requested video-mode, the position in the parameter list that is alloted to "RecommendedMode" could take any value whatsoever. In such cases, we use a dash, the symbol for the anonymous variable.

Readers will notice too, that predicates (**select_mode_aux()** and **assert_graphics_modes()** are examples) have a number of rules. The collection of rules for a single predicate are referred to as a procedure--thus, the collection of rules for the "**assert_graphics_mode()**" make up the procedure, "**assert_graphics_mode()**". Often, as the rules for this procedure are, the rules that make up a procedure are mutually exclusive. So are the rules that make up the "**select_mode_aux()**" procedure. This predicate is passed two variables, the "RecommendedMode" and integer, Choice, that represents the user's choice from the menu. The choices are numbered consecutively; so, uppose that the user chooses the seventh item on on the menu--then "**select_mode_aux()**" takes on the values "**select_mode_aux(RecommendedMode,7)**" where "RecommendedMode" is an integer value specifying one of the video modes. To understand how Prolog proceeds in selecting which rule within a procedure should be fired, we must consider the how Prolog proceeds with the task from a procedural, rather than declarative, point-of-view. Prolog goes through the list of rules that make the "**select_mode_aux()**" procedure one by one, and in the order that they are appear, trying to make each one true. Most commonly, the method that uses for trying to make a rule true relies on pattern matching.

Suppose that best highest video mode that your system supports is mode 18 (640x480x16) and that this is the mode that you decide to initialize. Then "**select_video_mode()**" sends "**select_video_mode_aux()**" an '18' as a the "RecommendedMode" and a '7' as the Choice--so the predicate is bound with values "**select_video_mode(18,7)**". Prolog then starts through the rules for "**select_video_mode_aux()**" one after the other. The first rule it encounters has "**select_mode_aux(_,1)"** for its head. Prolog attempts to match "**select_video_mode_aux(18,7)**" to "**select_video_mode_aux(_,1)**". The 18 in the first binding for the predicate will match with the anonymous variable in the second--any integer will. But '7' cannot match '1', so Prolog gives up on this rule and goes on to the next. (Not that consequence of this is that the order of the clauses effects the programme in Prolog, at least when more than one rule makes up a procedure). The next rule has "**select_mode_aux(RecommendedMode,1)"** for its head. Prolog tries to match

"**select_mode_aux(18,7)**" with "**select_mode_aux(RecommendedMode,1)**". Since "RecommendedMode" in the second case is bound to '18', Prolog can match '18' and "RecommendedMode"--they are identical values. However, Prolog cannot match '7' and '1', and so it continues to next rule, and the next, failing each time to find a match.

Finally it reaches the rule that has "**select_mode_aux(_,7)"** as its head. Prolog attemptes to match "**select_mode_aux(18,7)**" with this head. It is able to match '18' with the anonymous variable, since any integer will match with it. Moreover the '7''s in the two match--so we have a match!. So Prolog then go on to see if it can make the other conditions in the clause true. The conditions are

```
select_mode_aux(_,7):-
    mode18_avail(true),!,
    fg_setmode(18).
```

So it attempts to see if can the database predicate "mode18_avai()" --facts that were asserted when the graphics capabilities of the system were checked--matchs "**mode18_avail(true)**". For some reason, all the other checking gives me good results on my system, but this one, though it is analogous with the checking for all the other modes, does not. Anyway, suppose it did. Then it would find that it could match for "**mode18_avail(true)**" in the database. In this case, knowing that system supports mode 18, we want simply to go ahead and set the graphics mode to 18; we don't want to attempt to match any other rules for "**select_mode_aux()**" We want to say, "O.K., that's it, thats the mode I want, look no farther!" That is what the "**!**" ("cut") does. It prunes off any branches in the search path that might also be satisfied by "**select_mode_aux(18,7)**"--such as the rule that says that the video mode can't be said to mode 18--the rule that Prolog would go to if could not find a match to "**mode18_avail(true)**" in the database. And that is most (but not all) of what there is to Prolog's "cut" mechanism that so many have found so difficult. (It has been likened to the "**Goto**" of logic programming!)

If we attempt to compile and link our example programme, we find ourselves facing another hitch. This hitch is a little more troublesome than the others, perhaps but of nothing like the magnitude of difficulty people make it out to be. If you ask the IDE to compile and run the programmes, you get an error message at the first occurrence of a (PDC Prolog version) of a Fastgraph predicate. And if you try to make an '.exe' file, you'll get a linker error for every one of the Fastgraph predicates you have used. The trouble is that "profast.lib" is not being linked in. The PDC Integrated Development Environment does offer a menu item that allows you to specify what library you want linked in, but this method of linking in library code has never worked for me--perhaps this is why I have read so many despaing comments about PDC Prolog interface with 'C' being impossible to use.

The easiest way to link in the Fastgraph library and the Prolog bindings to it is to exit the Integrated Development Environment and link at the command line. There are a few little curves that you have to manoever, but their none too sharp.

It is possible to have the '.exe' file that results from the linking the Prolog '.obj' modules and the 'C' '.obj' modules and/or 'C' code from a library is to startup either as a Prolog programme, or as a 'C' programme, that is, to the Prolog system's initialization and startup code linked in or to have the 'C' system's initialization and start up routines linked in. The recommended method, however, since the release of PDC Prolog version 3.2 is to have the main Prolog programme constructed as a 'C' programme, i.e. to have it started by a 'C' **main()** function, which ensures that the normal 'C' initialization and startup code is appended to the '.exe' file by the linker. Prolog Development Center provides the code that must be linked to enable the '.exe' file to startup by the standard 'C' method that calls **main()**; **the main()** that is provided immediately calls the Prolog startup procedures. Prolog Development Center also provides, in files called "tcinit.asm" or "mscinit.asm" (for TurboC/BorlandC and Microsoft C

respectively) code that looks after memory management. These routines, and order in which they are linked together have changed a little with the release of PDC Prolog 3.3. Current practices are given below.

Since the release 3.30, PDC Prolog the optlinks overlay linker. It's a good linker, and so we might as well take advantage of it. My perferences for optlink switches are included in the command given below. The '/CO' switch is short-form for the also usable long-form '/CODEVIEW' and instructs the Optlinks linker to append information at the end of the '.exe' file concerning line number and symbol information that is used by such debuggers as CodeView, OptDebug, Periscope, and Turbo Debugger. In order for the linker to incorporate this information in the '.exe' file, it must be present in the '.obj' files, and so the Prolog compiler option specifying that line numbers should be generated must be enabled. The '/LI' switch, short for the also usable long-form '/LINENUMBERS specifies that line-number information is to be included in the '.map' file. The '/DET' is the short for the also usable '/DETAILEDMAP specifies that optlinks linker should produce detailed segment list information at the beginning of the '.map' file. The "fastgraf.inp" file is another response file, that contains a list of the files containing library code that should be linked the '.exe' file. It contents are given below. Assuming that you are using Borland C++ and the directory structure for Borland C++ produced by running 'install.exe.' (if you are using a compiler other than Borland C++, you would have to change the specification, identifying the path to the library for the 'C' module that contains the large library model and the name of the module) would issue the following command:

optlinks /CO /LI /DET tcfirst c:\borlandc\lib\c0l tcinit cmain <prolog.obj.modules> <c.obj.modules> [filename].sym,
[filename].exe,[filename].map,@fastgraf.inp

where

<prolog.obj.modules> is the list of prolog '.obj' that modules you want to link into the '.exe' file

<c.obj.modules> is the list of 'C' '.obj' modules that you want to link into the '.exe' file.

For example, if we were linking the 'initfast.obj' file generated from the initfast.pro module given below, we would issue the command:

optlinks /CO /LI /DET tcfirst c:\borlandc\lib\c0l tcinit cmain initfast initfast,initfast,initfast, @fastgraf.inp

You can automate this process by using a batch file. Whenerver you use Fastgraph, you instruct the Prolog compiler to place the '.obj' '.sym' and '.map' files in one specific directory Let us supposie you you ask the Prolog compiler to place the '.obj' and '.exe' files in the same directory in which you store 'optlinks.exe' (the Optlinks linker); you then could install the batch file given below, 'fg_link.bat' into the same directory, and link a series of up to four modules together by typing "fg_link" followed, on the same line, by up to four filenames. The DOS batch-file structure would allow you to include more file names, using the same '%n' convention that 'fg_link.bat' uses, but DOS prevents a command line having more than a certain number, and the command given above just about stretches this to the limit.

And that is all that there is to it. Utterly simple. Certainly, there's no need for the consternation sometimes expressed on the bulletin boards. Over the years, I have programmed in Fortran, BASIC, Forth, APL, 'C', LISP (actually Scheme) and Prolog. While all of these language have something to recommend them--readers might note a preference for languages that include arrays and lists as a basic data structure--, in no other language have I had the success that I have had with Prolog. Prolog is a very high-level language, and its power has

enabled me to write programmes to accomplish chores that I could in no other language--I simply couldn't afford the time. I hope some programmers who have been swayed by some of the nasty rumours that circulate will dismiss what they have heard and give Prolog a try. I have included a short reading list to help get you started.

Books on Turbo/PDC Prolog

Safaa H. Hashim and Philip Seyer, *Turbo Prolog: Advanced Programming Techniques* (Blue Ridge Summit, PA: Tab Books Inc, 1988). Hashim and Seyer carry PDC Prolog just about as far as you could ever want it to go. The book consists of advanced examples that illustrate how to create knowledge-based systems in Turbo/PDC Prolog and programming that illustrate concept acquisition. PDC Prolog has often been criticized for excluding the metalogical capacities that other Prolog's include. The criticism is a bit unfair, since PDC Prolog also comes with an interpreter that allows has metalogical capabilities, though it is, as would be expected, slower and dirtier than their Prolog compiler. But Hashim and Seyer make the criticism irrelevent anyway, showing you how to create predicates that enable you to pass predicates as variables to other predicates and to assert and retract rules from the database at run-time. Their method expands the capacity of PDC Prolog, but studying it makes you realize why you rarely want to rely on those capacities. (I do wish though that PDC Prolog included the "functor()" and "var()" predicates). Readers should not the PDC Prolog version 3.3x has the capability of taking predicates as variables. Nonetheless, the book is invaluable for countering some of the hoary old saws about PDC Prolog being a hopelessly non-standard Prolog.

Safaa H. Hashim, *Exploring Hypertext Programming: Writing Knowledge Representation and Problem-Solving Programs* (Blue Ridge Summit, PA: Windcrest Books Advanced Technology Series, 1990). Prolog is natural choice for Hypertext programming, and Safaa Hashim shows you how to do, using Turbo/PDC Prolog to the max. Great for getting a sense of just how powerful the language is.

Daniel H. Marcellus, *Expert Systems Programming in Turbo Prolog* (Englewood Cliffs, N.J.: Prentice-Hall, 1989). Marcellus book is extremely valuable as an aid for making the transition between the toy programming examples one find in most primers to the more complex programmes that programmers actually write. It is a little demanding in places--not because it is unclear, but because it takes you into some of the challenging aspects of programming in Prolog. The book is also valuable as a introduction to some methods for programming for artificial intelligence. I recommend it highly.

Mick McAllister, *Illustrated Turbo Prolog 2.0* (Plano,TX: Wordward Publishing Inc., 1989. An excellent, hand-holding introduction.

Herbert Schild, *Advanced Turbo Prolog Version 1.1* (Berkeley, CA: Borlnad Osborne McGraw-Hill, 1987). Somewhat dated, as it covers only release 1.1, but still useful for the sections on natural language processing, vision and pattern recognition, and machine learning.

Lee Teft, *Programming in Turbo Prolog-with an Introduction to Knowledge-Based Systems* (Englewood Cliffs, N.J.: Prentice-Hall, 1989). A good, classroom-type introduction to Turbo/PDC Prolog that goes a little more deeply into the language than many primers do. Very clear and sound.

Carl Townsend, *Advanced Techniques in Turbo Prolog* (Alameda, CA: Sybex Inc., 1986). Townsend is also the author of *Introduction to Turbo Prolog*, also published by Sybex, that I can also recommend. *Advanced Techniques* is especially important because it is really a toolkit of useful predicates for handling strings, dates and lists. His set of predicates for handling lists is among the most complete available.

Keith Weiskamp and Terry Hengl, *Artificial Intelligence Programming with Turbo Prolog* (New York, N.Y.: John Wiley and Sons, 1988). A useful introduction to programming for artificial intelligence, the book also includes an extremely valuable set of predicates for character and string handling.

Khin Maung Yin, with David Solomon, *Using Turbo Prolog* (Indianapolis, IN: Que Corporation, 1987). This book is dated, as it deals with Turbo Prolog version 1.1. Its material on using databases and disk files, though out of date (it doesn't cover B-Trees, for example) makes it useful nonetheless.

Books on other Prolog's (but relevent to Turbo Prolog)

Ramachandran Barath, *An Introduction to Prolog* (Blue Ridge Summit, PA: Tab Books Inc, 1986). An interesting approach, Barath takes you through a number of Prolog programmes step-by-step, using a question-and-answer approach. This can be a little tedious, but no other book gives you as vivid a sense of how Prolog computes.

Ramachandran Barath, *Prolog: Sophisticated Applications in Artificial Intelligence* (Blue Ridge Summit, PA: Windcrest Books, 1989). Covers the most common examples used for teaching Prolog, clearly and elegantly.

Ivan Bratko, Prolog: *Programming for Artificial Intelligence* (Wokingham, England: Addison-Wesley Publishing Company, 1990). A complete Prolog course, from basics up to advanced techniques. A special virtue of this book is its emphasis on data structures. Excellent--in my view next to Covington et. al. in general usefulness. Covington et. al. is faster-paced, Bratko more thorough in treatment of the orthodox computer science topics. One of the great strengths of Prolog, that I have not even mentioned, is the capacity to create recursive data structures. This is a wonderful capacity to have, and Prolog is the only mainstream language that supports. Bratko is good on the topic of recursive data structures.

W.F. Clocksin and C.S. Mellish, *Programming in PROLOG* 2nd ed. (Berlin: Springer-Verlag, 1984). The classic that established what "standard Prolog" means. Invaluable for discussion of accumulators, difference lists, "logical" applications of logic programming, etc. (but I didn't find it much fun).

Michael A. Covington, Donald Nute, André Vellino, *Prolog Programming in Depth* (Clearview, IL: Scott, Foresman and Company, 1988) A complete fast-paced introduction to Prolog that takes from the basics to parsing and knowledge-based systems. In my opinion, if there is one indispensible on Prolog, this is it. I would recommend reading it first.

Gregory L. Lazarev, *Why Prolog? Justifying Logic Programming for Practical Applications* (Englewood Cliffs N.J.: Prentice-Hall, 1989). A practical introduction to the language. The book is best at conveying how programming in Prolog differs from programming in procedural

languages and why the benefits of programming in Prolog are. Lazarev is also a good antidote for those who view Prolog as an language good only for artificial intelligence applications. He provides two excellent, and cogent, arguments for giving Prolog a try--first, that Prolog is versatile language that should not be restricted to expert systems programming, and secondly--and perhaps the most important of all--that, Prolog's declarative semantics and its solid mathematical foundations changed how he thinks about programming.

Leon Sterling and Ehud Shapiro, *The Art of Prolog: Advanced Programming Techniques* (Cambridge, MA: The MIT Press, 1986). The subtitle is a bit of misnomer--the book actually takes you from the ground up (though there are best choices for a primer). The book's particular strength isits stress on the concept of logic programming. It clearly presents where Prolog conforms to the idea of logic programming and where it departs from it.

The *Professional User's Guide* from the Prolog Development Center treats the topic of linking 'C' and Prolog in bone-grinding detail. It contains everything you might ever need to know about the topic--and perhaps more. (Don't say they didn't do their best in instructing us on the compatibility of 'C' and PDC Prolog. They also provide the files necessary to link Borland C '.obj' and Microsoft C '.obj' modules with PDC Prolog '.obj' modules.

Finally, there is one other article that I know of that presents the case for using PDC Prolog as a high level programming language and linking PDC Prolog modules with 'C' modules. That article is:

Gary Emtsming, "Prolog as a High-Level Toolbox" in *AIExpert* vol. 8, no. 3 (March 1991), pp. 56-61.

==========

```
/*-------------------------------------------------------------
The following are a set of 'C' modules that provide and interface between the Fastgraph library
for 'C' and PDC Prolog.  I've also included a few extra functions for good measure.  They should
be kept as separate modules and not bundled together, since the '.obj' modules we produce
from them are a going to be bundled together in a library.
-------------------------------------------------------------*/

/* abort.c */
/*-------------------------------------------------------------
void abort_program( int abort_code)
    called when the program can't be run
-------------------------------------------------------------*/

void fg_abort_program(int abort_code)
{
/* reset the mode to what it probably was --mode 3 */

    fg_setmode(3);
    fg_reset();

/* print a line that will stay on the screen after the program exits */
```

```
    if (abort_code == 0)
       {
          printf("\nYour system does not have enough memory");
          printf("\n to run this program.\n");
       }
     else if (abort_code == 1)
         printf("\nThe font file is missing.\n");
    exit(0);
}

/* alloca_p.c */

void fg_allocate_0(int page, int *out)
    {
    extern int fg_allocate(int);

    *out = fg_allocate(page);
    }

/*  allocc_p.c */

void  fg_alloccms_0(int page,int *success)
{
    int fg_alloccms(int);

    *success = fg_alloccms(page);
}
/* Note this file is not called alloccms.c because then the name of the object file would conflict
with one that Gruber has already stored in the Fastgraph Library. */

/*  alloce_p.c */

void  fg_allocems_0(int page,int *success)
{
    int fg_alloecms(int);

    *success = fg_allocems(page);
}

/*  allocx_p.c */
void  fg_allocxms_0(int page,int *success)
{
    int fg_alloxcms(int);

    *success = fg_allocxms(page);
}

/* arryli_p.c */

#define listfno  1
```

```
#define nilfno   2

void *MEM_AllocGStack(unsigned);

typedef struct ilist {
   char Functor;
   int Value;
   struct ilist *Next;
} IntList;

typedef struct rlist {
   char Functor;
   int Value;
   struct ilist *Next;
} RealList;


/*-------------------------------------------------------------
converting between lists and arrays of integers
This code is taken from the PDC Prolog Reference Guide,
release 3.2, pg 98, release 3.3, pp. 103-104.)
-------------------------------------------------------------*/

int IntListToArray(IntList *List,int **ResultArray)
{   /* Convert a list to an array placed on the global stack */
   IntList *SaveList = List;
   int *Array;
   int  i = 0;

 /* Count the number of elements in the list */
   for(i=0; List->Functor==listfno; List=List->Next)
      i++;

    Array = MEM_AllocGStack(i*sizeof(int));
/* Allocate the needed memory */
          List = SaveList;
/* Transfer the elements from the list to the array */

   for(i=0; List->Functor==listfno; List=List->Next)
      Array[i++]=List->Value;

   *ResultArray=Array;
   return(i);
}
void IntArrayToList(int Array[],int n,IntList **List)
 /* Convert an array to a list */
{
   int i;

   for (i=0; i<n; i++)  /* Allocate a record for each element */
    {
```

```
    IntList *p=*List=MEM_AllocGStack(sizeof(IntList));
    p->Functor=listfno;
    p->Value=Array[i];
    List=&(*List)->Next;
  }

  {  /* Allocate the last record in the list */
    IntList *p=*List=MEM_AllocGStack(sizeof(char));
    p->Functor=nilfno;
  }
}


/*-------------------------------------------------------------
For converting between lists and arrays of reals. Based on
code written by the Prolog Development Center, and published in the PDC Prolog Reference
Manual (release 3.2, pg 98, release 3.3, pp 103-104.)
-------------------------------------------------------------*/

int RealListToArray(RealList *List, double **ResultArray)
{
 /* Convert a list to an array placed on the global stack */
    RealList *SaveList = List;
    double *Array;
    int  i = 0;

 /* Count the number of elements in the list */
  for(i=0; List->Functor==listfno; List=List->Next)
     i++;

  Array = MEM_AllocGStack(i*sizeof(double));
  /* Allocate the needed memory */
  List = SaveList;

  /* Transfer the elements from the list to the array */

  for(i=0; %GMK%List->Functor==listfno; List=List->Next)
    Array[i++]=List->Value;

  *ResultArray=Array;
  return(i);
}

void RealArrayToList(double Array[],int n,RealList **List)
 /* Convert an array to a list */
{
  int i;
  for (i=0; i<n; i++)  /* Allocate a record for each element */
   {
    RealList *p=*List=MEM_AllocGStack(sizeof(RealList));
    p->Functor=listfno;
```

```
    p->Value=Array[i];
    List=&(*List)->Next;
  }

  {  /* Allocate the last record in the list */
    RealList *p=*List=MEM_AllocGStack(sizeof(char));
    p->Functor=nilfno;
  }
}


int IntListToByteArray(IntList *List,int **ResultArray)
{   /* Convert a list to an array placed on the global stack */
  IntList *SaveList = List;
  int *Array;
  int  i = 0;

 /* Count the number of elements in the list */
  for(i=0; List->Functor==listfno; List=List->Next)
    i++;

   Array = MEM_AllocGStack(i*sizeof(int));

/* Allocate the needed memory */
  List = SaveList;
/* Transfer the elements from the list to the array */

  for(i=0; List->Functor==listfno; List=List->Next)
    Array[i++]= (char) List->Value;

  *ResultArray=Array;
  return(i);
}

void ByteArrayToIntList(int Array[],int n,IntList **List)
 /* Convert an array to a list */
{
  int i;

  for (i=0; i<n; i++)  /* Allocate a record for each element */
   {
    IntList *p=*List=MEM_AllocGStack(sizeof(IntList));
    p->Functor=listfno;
    p->Value=(int)Array[i];
    List=&(*List)->Next;
  }

  {  /* Allocate the last record in the list */
    IntList *p=*List=MEM_AllocGStack(sizeof(char));
    p->Functor=nilfno;
  }
```

```
}

/* automo_p.c */

extern int fg_autoload(void);

void fg_automode_0( int * out)
    {
    *out = fg_autoload();
    }


/* bestmo_p.c */

void fg_bestmode_0(int horiz,int vert,int page,int * mode)
     {
      extern int fg_bestmode(int, int, int);

      *mode = fg_bestmode(horiz, vert, page);
    }

/* button_p.c */

void fg_button_0(int n, int *state)
    {
    extern int fg_button(int);

    *state = fg_button(n);
    }

/* capslo_p.c */

void fg_capslock_0(int *state)
    {
    extern int fg_capslock(void);

    *state = fg_capslock();
    }

/* disppc_p.c */


void fg_disppcx_0(char *fname,int mode,int *status)
    {
    extern int fg_disppcx(char *fname,int mode);

    *status = fg_disppcx(fname,mode);
     }

/*dispprf.c  */
```

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<io.h>
#include"fastgraf.h"
// "fastgraf.h" is part of Ted Gruber Software Fastgraph.
// Fastgraph is available as a shareware
//  progamme on many bulletins boards, or from the
//  DustDevil BBS, Los Vegas, Nevada (702) 796-7134)
//  as part of the fastgraph/light
//  shareware package.

#include<stdio.h>
#include<mem.h>
/*-------------------------------------------------------------
void fg_dispprf_0(char *fname, int* return_status)
void fg_getprfsize
          (char *filename,int *width,int *len,int
          format,int*return_status)
Code written by John Wagner, author of the fine shareware programme 'Improcess'--my
favourite image processing package
with which I have done many of the special effects for my
recent film animations. The '.prf' file format for images was set by Ted Gruber of Ted Gruber
software.
-------------------------------------------------------------*/

int prf_size(char *filename, int *width,
           int *length, int *prf_format)

{
   FILE *fp;
   unsigned char fgheader[26];
   int return_value;
   fp = fopen(filename,"rb");
   if(fp != NULL)
    {
    if(fread(fgheader, 26, 1, fp) == 1)
     {
      if(!memcmp(fgheader,"F\0A\0S\0T\0G\0R\0A\0F\0",16))
     {
          *width = fgheader[18];
          (*width) *= 256;
          (*width) +=fgheader[16];
          (*width)++;
          if(length)
            {
            *length = fgheader[22];
            (*length) *= 256;
            (*length) +=fgheader[20];
            (*length)++;
```

```
            }
           *prf_format = fgheader[24];
           return_value = 1;
          }
          else return_value = 0;
     }
     else return_value = 0;
     fclose(fp);
    }
    else
    {
      *width = 0;
      *length = 0;
      return_value = -1;
     }
 return(return_value);
}

int prf_size(char *filename,int *width,
           int *length,int *prf_format);

void fg_getprfsize_0(char *filename, int *width, int *length,
                  int *prf_format,int *return_value)
{

    *return_value = prf_size(filename, width,length,prf_format); }

void fg_dispprf_0(char* fname, int* return_status)
{
    int dispprf(char* fname);

    *return_status = dispprf(fname);

}

/* egache_p.c */

void fg_egacheck_0(int *no_of_bytes)
    {
    extern int fg_egacheck(void);

    *no_of_bytes = fg_egacheck();
    }

/* ems_pro.c */

void fg_initems_0(int *success)
{
    int fg_initems(void);

    *success =  fg_initems();
```

```
}


/* fexists_p.c */

/*-------------------------------------------------------------
    fg_exists(char* filename)     does the file exist?
-------------------------------------------------------------*/

int fg_fexists(char* filename)
{
    if (access(filename,0) == 0) return(1);
    else  return(0);
}

void fg_fexists_0(char* filename, int * result)
{
    int fg_fexists(char* );

    *result = fg_fexists(filename);
}

/* freepa_p.c */

void fg_freepage_0(int page_number,int *report)
    {
    extern int fg_freepage(int);

    *report = fg_freepage(page_number);
    }

/* getadd_p.c */

void fg_addr_0(int * addr_of_active_page)
    {
    extern int fg_getaddr(void);

    *addr_of_active_page = fg_getaddr();
    }

/* getclo_p.c */

void fg_getclock_0(float * num_of_ticks)
    {
    extern long fg_getclock(void);

    *num_of_ticks = (float) fg_getclock();
    }

/* getcol_p.c */
```

```
void fg_getcolor_0(int * color)
    {
    extern int fg_getcolor(void);

    *color = fg_getcolor();
    }

/* palettes.c */

#define listfno  1
#define nilfno   2
void *MEM_AllocGStack(unsigned);

typedef struct ilist {
   char Functor;
   int Value;
   struct ilist *Next;
} IntList;


void fg_getdacs_0(int start, int count, IntList ** values)
    {
    char * dacregisters;
    extern void fg_getdacs(int, int, char *);

    fg_getdacs(start,count,dacregisters);
    ByteArrayToIntList(dacregisters,count,values);
    }

/* gethpa_p.c  */

void fg_gethpage_0(int *page)
    {
    extern int fg_gethpage(void);

    *page = fg_gethpage();
    }


/* getind_p.c */

void fg_getindex_0(int index, int * colour_val)
    {
    extern int fg_getindex(int);

    *colour_val = fg_getindex(index);
    }

/*  getkey_p.c */

void fg_getkey_0(int *keyint, int *auxint)
```

```
{
     char *key, *aux;
     void fg_getkey(unsigned char*, unsigned char*);


     fg_getkey(key, aux);
     *keyint = (int) *key;
     *auxint = (int) * aux;
}

/* getmax_p.c */

void fg_getmaxx_0(int * maxx)
    {
    extern int fg_getmaxx(void);

    *maxx = fg_getmaxx();
    }

/* getmay_p.c */

void fg_getmaxy_0(int * maxy)
    {
    extern int fg_getmaxy(void);

    *maxy = fg_getmaxy();
    }

/* getmod_p.c */

void fg_getmode_0(int * mode)
    {
    extern int fg_getmode(void);

    *mode = fg_getmode();
    }

/* getpag_p.c */

void fg_getpage_0(int * page)
    {
    extern int fg_getpage(void);

    *page = fg_getpage();
    }

/* getpix_p.c */

void fg_getpixel_0(int x, int y, int * colour )
    {
    extern int fg_getpixel(int,int);
```

```
    *colour = fg_getpixel(x, y);
    }

/* getvpa_p.c */

void fg_getvpage_0(int * vpage_num)
    {
    extern int fg_getvpage(void);

    *vpage_num = fg_getvpage();
    }

/* getxjo_p.x */

void fg_getxjoy_0(int n, int * x_coord)
    {
    extern int fg_getxjoy(int);

    *x_coord = fg_getxjoy(n);
    }

/* getxpo_p.c */

void fg_getxpos_0(int * xpos)
    {
    extern int fg_getxpos(void);

    *xpos = fg_getxpos();
    }

/* getyjo_p.x */

void fg_getyjoy_0(int n, int * y_coord)
    {
    extern int fg_getyjoy(int);

    *y_coord = fg_getyjoy(n);
    }

/* getxpo_p.c */

void fg_getypos_0(int * ypos)
    {
    extern int fg_getypos(void);

    *ypos = fg_getypos();
    }

/* initjo_p.c */
```

```
void fg_initjoy_0(int n, int * report)
    {
    extern int fg_initjoy(int);

    *report = fg_initjoy(n);
    }

/* makepc_p.c */

void fg_makepcx_0(int xmin,int xmax,int ymin,int ymax,
                char *fname,int *status)
     {
    extern int fg_makepcx(int,int,int,int,char*);

    *status = fg_makepcx(xmin,xmax,ymin,ymax,fname);
    }

/* MAPRGB_p.C */

void fg_maprgb_0(int red, int green, int blue, int * val)
{
    extern int fg_maprgb(int,int,int);

    *val = fg_maprgb(red,green,blue);
}

/* measur_p.c */

void fg_measure_0(int * measure )
    {
    extern int fg_measure(void);

    *measure = fg_measure();
    }

/* memava_p.c */

void fg_memavail_0(long * memavail)
    {
    extern long fg_memavail(void);

    *memavail = fg_memavail();
    }

/* memavail.c */

void fg_memavail_0(long * memavail)
    {
    extern long fg_memavail(void);

    *memavail = fg_memavail();
```

```
    }

/* mousei_p.c */

void fg_mouseini_0(int * numbuttons)
    {
    extern int fg_mouseini(void);

    *numbuttons = fg_mouseini();
    }

/* numloc_p.c  */

void fg_numlock_0(int * state)
    {
    extern int fg_numlock(void);

    *state = fg_numlock();
    }

/* palett_p.c */

#define listfno  1
#define nilfno   2

void *MEM_AllocGStack(unsigned);

typedef struct ilist {
   char Functor;
   int Value;
   struct ilist *Next;
} IntList;


extern int IntListToArray(IntList *List,int **ResultArray);
extern void IntArrayToList(int Array[],int n,IntList **List);

void fg_palettes_0(IntList **colour_list)
    {
    int *colour_array;
    extern void fg_palettes(int *);

    fg_palettes(colour_array);
    IntArrayToList(colour_array, 16, colour_list);
    }

/* pcx_size.c  */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include<mem.h>
#include<io.h>
#include "c:\fastgraf\fastgraf.h"
#include "c:\prolog\cprogs\pdcrunt.h"
 /* this file is part of the PDCPackage--it is composed of
     declarations for the functions in prolog.lib */


/*-------------------------------------------------------------
    int pcx_size(char *filename,int *width,int *length
-------------------------------------------------------------*/

typedef struct {
        char manufacturer;
        char version;
        char encoding;
        char bits_per_pixel;
        int xmin,ymin;
        int xmax,ymax;
        int hres;
        int vres;
        char palette[48];
        char reserved;
        char colour_planes;
        int bytes_per_line;
        int palette_type;
        char filler[58];
        }  PCXHEAD;


PCXHEAD header;
unsigned int width, depth;
unsigned int bytes;


int fg_pcxsize(char *filename,int *width,int *length)
{
    FILE *fp;
    int return_value;


    fp = fopen(filename,"rb");
    if(fp != NULL)
     {
     if(fread( (char *)&header,1,sizeof(PCXHEAD),fp) ==
                                                                    sizeof(PCXHEAD))
       {
        if( header.manufacturer == 0x0a )
            {
             *width = (header.xmax-header.xmin) + 1;
             *length = (header.ymax-header.ymin) + 1;
```

```
            return_value = 1;
          }
          else return_value = 7;
      }
      else return_value = 0;
      fclose(fp);
     }
     else
     {
       *width = 0;
       *length = 0;
       return_value=-1;
      }
 return(return_value);
}


void  fg_pcxsize_0(char* filename,int* width, int *length,
                int *rstatus)
 {
     int fg_pcxsize(char *,int *,int * );

     *rstatus = fg_pcxsize(filename,width,length);
}

/* playin_p.c  */

void fg_playing_0(int * boole)
     {
     extern int fg_playing(void);

     *boole = fg_playing();
     }

/* polygo_p.c */
#define listfno  1
#define nilfno   2

void *MEM_AllocGStack(unsigned);

typedef struct ilist {
   char Functor;
   int Value;
   struct ilist *Next;
} IntList;


extern int IntListToArray(IntList *List,int **ResultArray);
extern void IntArrayToList(int Array[],int n,IntList **List);

void fg_polygon_0(IntList *xlist, IntList *ylist)
```

```
{
    int *xarray, *yarray, n =0, m = 1;
    extern void fg_polygon(int *, int *, int);

    n = IntListtoArray(xlist,&xarray);
    m = IntListToArray(ylist,&yarray);

    if (m != n)
    {
        printf("\nUnequal number of x and y coordinates in");
        printf("\nfor polygon.");
        exit(0);
    }
    else
    fg_polygon(xarray,yarray,n);
}


/* polygw_p.c */

#define listfno  1
#define nilfno   2

void *MEM_AllocGStack(unsigned);

typedef struct rlist {
  char Functor;
  int Value;
  struct rlist *Next;
} RealList;

extern int RealListToArray(RealList *List, double **ResultArray);
extern void RealArrayToList(double Array[],int n,
                        RealList **List);

void fg_polygonw_0(RealList *xlist, RealList *ylist)
{
    double *xarray, *yarray, n =0, m = 1;
    extern void fg_polygonw(double *, double *, int);

    n = RealListtoArray(xlist,&xarray);
    m = RealListToArray(ylist,&yarray);

    if (m != n)
    {
        printf("\nUnequal number of x and y coordinates in");
        printf("\nfor polygon.");
        exit(0);
    }
    else
    fg_polygonw(xarray,yarray,n);
```

```
}

/* prfsize.c */

/*-------------------------------------------------------------
A programme by John Wagner, author of Improces, an extraordinary shareware package for
image processing. '.prf'  files are image files produced by Fastgraph.  The '.prf' file format is the
work of Ted Gruber of Ted Gruber software, the creators of Fastgraph.
-------------------------------------------------------------*/
#include<stdio.h>
#include<mem.h>

int fg_prfsize(char *filename, int *width, int *length,
                               int *prf_format)
{
   FILE *fp;
   unsigned char fgheader[26];
   int return_value;
   fp = fopen(filename,"rb");
   if(fp != NULL)
    {
    if(fread(fgheader, 26, 1, fp) == 1)
     {
       if(!memcmp(fgheader,"F\0A\0S\0T\0G\0R\0A\0F\0",16))            {
            *width = fgheader[18];
            (*width) *= 256;
            (*width) +=fgheader[16];
            (*width)++;
            if(length)
              {
               *length = fgheader[22];
               (*length) *= 256;
               (*length) +=fgheader[20];
               (*length)++;
                }
            *prf_format = fgheader[24];
            return_value = 1;
           }
           else return_value = 0;
     }
     else return_value = 0;
     fclose(fp);
    }
    else
    {
       *width = 0;
       *length = 0;
       return_value=-1;
     }
 return(return_value);
}
```

```
fg_prfsize_0(char *filename,int *width,int *length,
            int *prf_format,int * status)
     {
     int fg_prfsize( char*, int*, int *, int *);

     *status = fg_prfsize(filename,width,length,prf_format);
      }

/* scrloc_p.c */

void fg_scrlock_0(int *state)
     {
     extern int fg_scrlock(void);

     *state = fg_scrlock();

     }

/* setdac_p.c */

#define listfno  1
#define nilfno   2

void *MEM_AllocGStack(unsigned);

typedef struct ilist {
   char Functor;
   int Value;
   struct ilist *Next;
} IntList;


extern int IntListToCharArray(IntList *List,char **ResultArray);

void fg_setdacs_0(int start, int count, IntList *valslist)
     {
     char *dacs_array;
     extern void fg_setdacs(int, int, char *);

     IntListToCharArray(valslist,&dacs_array);
     fg_setdacs(start, count, dacs_array);
     }


/* sounds_p.c */

#define listfno  1
#define nilfno   2

void *MEM_AllocGStack(unsigned);
```

```
typedef struct ilist {
   char Functor;
   int Value;
   struct ilist *Next;
} IntList;

extern int IntListToArray(IntList *List,int **ResultArray);
extern void IntArrayToList(int Array[],int n,IntList **List);

void fg_sounds_0(IntList *sounds, int ntimes)
    {
    int *sounds_array;
    extern void fg_sounds(int *, int);

    IntListToArray(sounds,&sounds_array);
    fg_sounds(sounds_array, ntimes);
    }

/*  swleng_p.c  */

void fg_swlength_0(char *string, int n, double * length)
    {
    extern double fg_swlength(char *, int);

    *length = fg_swlength(string, n);
    }

/* testmo_p.c */

void fg_testmode_0(int mode, int pages, int *boole)
    {
    extern int fg_testmode(int,int);

    *boole = fg_testmode(mode,pages);
    }

/* xalpha_p.c */

void fg_xalpha_0(int screen_space_x, int * char_space_col)
    {
    extern int fg_xalpha(int);

    *char_space_col = fg_xalpha(screen_space_x);
    }

/* xconver_p.c */

void fg_xconvert_0(int column, int * left_pixel)
    {
    extern int fg_xconvert(int);
```

```
    *left_pixel = fg_xconvert(column);
    }

/* xms_pro.c */

void fg_initxms_0(int *success)
{
    int fg_initxms(void);

    *success =  fg_initxms();
}

/* xscreen_p.c */

void fg_xscreen_0(double world_space_x, int * screen_space_x)
     {
    extern int fg_xscreen(double);

    *screen_space_x = fg_xscreen(world_space_x);
    }

/* xworld_p.c */

void fg_xworld_0(int screen_space_x, double * world_space_x)
     {
     extern double fg_xworld(int);

    *world_space_x = fg_xworld(screen_space_x);
    }

/* yalpha_p.c */

void fg_yalpha_0(int screen_space_y, int * char_space_row)
    {
    extern int fg_yalpha(int);

    *char_space_row = fg_yalpha(screen_space_y);
    }

/* yconver_p.c */

void fg_yconvert_0(int column, int * top_pixel)
    {
    extern int fg_yconvert(int);

    *top_pixel = fg_yconvert(column);
    }

/* yscree_p.c */
```

```
void fg_yscreen_0(double world_space_y, int * screen_space_y)
     {
      extern int fg_yscreen(double);

     *screen_space_y = fg_xscreen(world_space_y);
     }

/* yworld_p.c */

void fg_yworld_0(int screen_space_y, double * world_space_y)
     {
     extern double fg_yworld(int);

     *world_space_y = fg_xworld(screen_space_y);
     }

/*-------------------------------------------------------------
FASTGRAF.PRE                                    22/01/92
                                REV.  R.B.E. 29/07/92
  predicates in fastgraf.lib (rewritten for PDC Prolog)
-------------------------------------------------------------*/

GLOBAL DOMAINS

reallist = real*
block = binary

GLOBAL PREDICATES

/* fastgraph predicates in Grubers library or written in c */

determ fg_abort_program(integer) - (i) language c    as
                                "_fg_abort_program"
determ fg_allocate(integer,integer) - (i,o)  language c as
                                "_fg_allocate_0"
determ fg_alloccms(integer,integer) - (i,o)  language c as
                                "_fg_alloccms_0"
determ fg_allocems(integer,integer) - (i,o)  language c as
                                "_fg_allocems_0"
determ fg_allocxms(integer,integer) - (i,o)  language c as
                                "_fg_allocxms_0"
determ fg_automode(integer) - (o) language c as "_fg_automode_0"
determ fg_bestmode(integer,integer,integer,integer) - (i,i,i,o)
                                 language c as "_fg_bestmode_0"
determ fg_button(integer,integer) - (i,o) language c as
                                "_fg_button_0"
determ fg_capslock(integer) - (o)  language c as "_fg_caplock_0"
determ fg_click(integer) - (i)  language c as "_fg_click_0"
determ fg_disppcx(string,integer,integer) - (i,i,o)   language c
                                as "_fg_disppcx_0"
determ fg_dispprf(string,integer) - (i,o) language c as
```

```
                        "_fg_dispprf_0"
determ fg_egacheck(integer) - (o) language c as "_fg_egacheck_0"
determ fg_fexists(string, integer) - (i,o) language c as
                        "_fg_fexists_0"
determ fg_freepage(integer,integer) - (i,o) language c as
                        "_fg_freepage_0"
determ fg_getaddr(integer) - (o) language c as "_fg_getaddr_0"
determ fg_getclock(real) - (o) language c as "_fg_getclock_0"
determ fg_getcolor(integer) - (o) language c as "_fg_getcolor_0"
determ fg_getcolour(integer) - (o) language c as "_fg_getcolor_0"
determ fg_gethpage(integer) - (o) language c as "_fg_gethpage_0"
determ fg_getindex(integer,integer) - (i,o) language c as
                        "_fg_getindex_0"
determ fg_getkey(integer,integer) - (o,o) language c as
                        "_fg_getkey_0"
determ fg_getmaxx(integer) - (o) language c as "_fg_getmaxx_0"
determ fg_getmaxy(integer) - (o) language c as "_fg_getmaxy_0"
determ fg_getmode(integer) - (o) language c as "_fg_getmode_0"
 determ fg_getpage(integer) - (o) language c as "_fg_getpage_0"
determ fg_getpixel(integer,integer,integer) - (i,i,o) language c
                        as "_fg_getpixel_0"
determ fg_getvpage(integer) - (o) language c as "_fg_getvpage_0"
determ fg_getxjoy(integer,integer) - (i,o) language c as
                        "_fg_getxjoy"
determ fg_getxpos(integer) - (o) language c as "_fg_getxpos_0"
determ fg_getyjoy(integer,integer) - (i,o) language c as
                        "_fg_getyjoy_0"
determ fg_getypos(integer) - (o) language c as "_fg_getypos_0"
determ fg_initems(integer) - (o) language c as "_fg_initems_0"
determ fg_initjoy(integer,integer) - (i,o) language c as
                        "_fg_initjoy_0"
determ fg_initxms(integer) - (o) language c as "_fg_initxms_0"
 determ fg_length_pstring(string,integer) - (i,o) language c as
                        "_fg_length_pstring_0"
determ fg_makepcx(integer,integer,integer,integer,string,integer)
                        - (i,i,i,i,i,o)  language c as "_fg_makepcx_0"
determ fg_maprgb(integer,integer,integer,integer) - (i,i,i,o)
                        langugage c as "_fg_maprgb_0"
determ fg_measure(integer) - (o) language c as "_fg_measure_0"
determ fg_numlock(integer) - (o) language c as "_fg_numlock_0"
determ fg_mouseini(integer) - (o) language c as "_fg_mouseini_0"
determ fg_pcxsize(string,integer,integer,integer) - (i,o,o,o)
                         language c as "_fg_pcxsize_0"
determ fg_pcx_colours(string,integer,integer) - (i,o,o) language
c as "_fg_pcx_colours_0"
determ fg_pcx_data(string,integer,integer) - (i,o,o) language c
                        as "_fg_pcx_data_0"
determ fg_playing(integer) - (o) language c as "_fg_playing_0" determ
fg_prfsize(string,integer,integer,integer,integer) -
                        (i,o,o,o,o)  language c as "_fg_prfsize_0"
determ fg_row_offset(integer, integer) - (i,o) language c as
```

```
                              "_fg_row_offset_0"
determ fg_scrlock(integer) - (o) language c as "_fg_scrlock_0"
 determ fg_xalpha(integer,integer) - (i,o) language c as
                              "_fg_xalpha_0"
determ fg_xconvert(integer,integer) - (i,o) language c as
                               "_fg_xconvert_0"
determ fg_xscreen(real,integer) - (i,o) language c as
                              "_fg_xscreen_0"
determ fg_xworld(integer,real) - (i,o)  language c as
                                                    "_fg_xworld_0"
determ fg_yalpha(integer,integer) - (i,o) language c as
                              "_fg_yalpha_0"
determ fg_yconvert(integer,integer) - (i,i) language c as
                              "_fg_yconvert_0"
determ fg_yscreen(real,integer) - (i,o) language c as
                              "_fg_yscreen_0"
determ fg_yworld(real,integer) - (i,o) language c as                    "_fg_yworld_0"
determ fg_swlength(string,integer,real) - (i,i,o) language c as
                              "_fg_swlength_0"
determ fg_testmode(integer,integer,integer) - (i,i,o) language c
                              as "_fg_testmode_0"
determ fg_palettes(integerlist) - (i) language c                        as
 "_fg_palettes_0"
determ fg_sounds(integerlist,integer) - (i,i) language c as
                              "_fg_sounds_0"
determ fg_voices(integerlist,integer) - (i,i) language c as
                              "_fg_voices_0"
determ fg_polygon(integerlist,integerlist,integer) - (i,i,o)
                              language c as "_fg_polygon_0"
determ fg_polygonw(reallist,reallist,integer) - (i,i,i) language
                              c as "_fg_polygonw_0"
determ fg_mouseptr(integerlist,integer,integer) - (i,i,i)
                              language c as "_fg_mouseptr_0"
determ fg_chgattr(integer) - (i) language c as "_fg_chgattr"
determ fg_circle(integer)  - (i) language c as "_fg_circle"
determ fg_circlew(real)    - (i) language c as "_fg_circlew"
determ fg_clipmask(string,integer,integer) - (i,i,i) language c
                              as "_fg_clipmask"
determ fg_clpimage(string,integer, integer) - (i,i,i) language c
                              as "_fg_clipimage"
determ fg_clprect(integer,integer,integer,integer) - (i,i,i,i)
                              language c as "_fg_clprect"
determ fg_clprectw(real,real,real,real) - (i,i,i,i) language c as
                              "_fg_clprectw"
determ fg_copypage(integer,integer) - (i,i) language c as
                              "_fg_copypage"
determ fg_cursor(integer) - (i) language c as "_fg_cursor"
determ fg_dash(integer,integer,integer) - (i,i,i) language c as
                              "_fg_dash"
determ fg_dashrel(integer,integer,integer) - (i,i,i) language c
                              as "_fg_dashrel"
```

```
determ fg_dashrw(real,real,integer) - (i,i,i) language c as
                        "_fg_dashrw"
determ fg_dashw(real,real,integer) - (i,i,i) language c as
                        "_fg_dashw"
determ fg_defcolor(integer,integer) - (i,i) language c as
                        "_fg_defcolor"
determ fg_dispfile(string,integer,integer) - (i,i,i) language c
                        as "_fg_dispfile"
determ fg_display(string,integer,integer) - (i,i,i) language c as
                        "_fg_display"
determ fg_displayp(string,integer,integer) - (i,i,i) language c
                        as "_fg_displayp"
determ fg_draw(integer,integer) - (i,i) language c as "_fg_draw"
 determ fg_drawmap(string,integer,integer) - (i,i,i) language c
 as                     "_fg_drawmap"
determ fg_drawmask(string,integer,integer) - (i,i,i) language c
                        as "_fg_drawmask"
determ fg_drawrel(integer,integer) - (i,i) language c as
                        "_fg_drawrel"
determ fg_drawrw(real,real) - (i,i) language c as "_fg_drawrw"
determ fg_draww(real,real) - (i,i) language c as "_fg_drawrw"
determ fg_drect(integer,integer,integer,integer,string) -
                        (i,i,i,i,i) language c as "_fg_drect"
determ fg_drectw(real,real,real,real,string) - (i,i,i,i,i)
                        language c as "_fg_drectw"
determ fg_drwimage(string,integer,integer) - (i,i,i) language c
                        as "_fg_drimage"
determ fg_ellipse(integer,integer) - (i,i) language c as
                        "_fg_ellipse"
determ fg_ellipsew(real,real) - (i,i) language c as
                        "_fg_ellipsew"
determ fg_erase - language c as "_fg_erase"
determ fg_erase_char(integer,integer) - (i,i) language c as
                        "_fg_erase_char"
determ fg_fadein(integer) - (i) language c as "_fg_fadein"
determ fg_fadeout(integer) - (i) language c as "_fg_fadeout"
 determ fg_flipmask(string,integer,integer) - (i,i,i) language c
                        as "_fg_flipmask"
determ fg_flpimage(string,integer,integer) - (i,i,i) language c
                        as "_fg_flpimage"
determ fg_getdacs(integer,integer,integerlist) - (i,i,o) language
                        c as "_fg_getdacs_0"
determ fg_getimage(block,integer,integer) - (o,i,i) language c as
                        "_fg_getimage"
determ fg_getmap(block,integer,integer) -(o,i,i) language c as
                        "_fg_getmap"
determ fg_getrgb(integer,integer,integer,integer) - (i,o,o,o)
                        language c as "_fg_getrgb" determ fg_getworld(real,real,real,real) -
(o,o,o,o) language c as
                        "_fg_getworld"
determ fg_hush - language c as "_fg_hush"
```

```
determ fg_hushnext - language c as "_fg_hushnext"
determ fg_initw - language c as "_fg_initw"
determ fg_intjoy(integer,char,char) - (i,o,o) language c as
                        "_fg_intjoy"
determ fg_intkey(char,char) - (o,o) language c as "_fg_intkey"
determ fg_locate(integer,integer) - (i,i) language c as
                        "_fg_locate"
determ fg_mousebut(integer,integer,integer,integer) - (i,o,o,o)
                        language c as "_fg_mousebut"
determ fg_mousecur(integer,integer) - (i,i) language c as
                        "_fg_mousecur"
determ fg_mouselim(integer,integer,integer,integer) - (i,i,i,i)
                        language c as "_fg_mouselim"
determ fg_mousemov(integer,integer) - (i,i) language c as
                        "_fg_mousemov"
determ fg_mousepos(integer,integer,integer) - (o,o,o) language c
                        as "_fg_mousepos"
determ fg_mousespd(integer,integer) - (i,i) language c as
                        "_fg_mousepd"
determ fg_mousevis(integer) - (i) language c as "_fg_mousevis"
determ fg_move(integer,integer) - (i,i) language c as "_fg_move"
determ fg_moverel(integer,integer) - (i,i) language c
                        as "_fg_moverw"
determ fg_moverw(real,real) - (i,i) language c as "_fg_moverw"
determ fg_movew(real,real) - (i,i) language c as "_fg_movew"
determ fg_music(string) - (i) language c as "_fg_music"
determ fg_musicb(string,integer) - (i,i) language c as
                        "_fg_musicb"
determ fg_paint(integer,integer) - (i,i) language c as
                         "_fg_paint"
determ fg_paintw(real,real) - (i,i) language c as "_fg_paintw"
determ fg_palette(integer,integer) - (i,i) language c as
                         "_fg_palette"
determ fg_pan(integer,integer) - (i,i) language c as "_fg_pan"
 determ fg_panw(real,real) - (i,i) language c as "_fg_panw"
determ fg_pattern(integer,integer) - (i,i) language c as
                        "_fg_pattern"
determ fg_point(integer,integer) - (i,i) language c as
                        "_fg_point"
determ fg_pointw(real,real) - (i,i) language c as "_fg_pointw"
 determ fg_quiet - language c as "_fg_quiet"
determ fg_rect(integer,integer,integer,integer) - (i,i,i,i)
                        language c as "_fg_rect"
determ fg_rectw(real,real,real,real) - (i,i,i,i) language c as
                        "_fg_rectw"
determ fg_reset - language c as "_fg_reset"
determ fg_restore(integer,integer,integer,integer) - (i,i,i,i)
 language c as "_fg_restore"
determ fg_restorew(real,real,real,real) - (i,i,i,i) language c as
                        "_fg_restorew"
determ fg_revimage(string,integer,integer) - (i,i,i) language c
```

```
                              as "_fg_revimage" determ
 fg_revmask(string,integer,integer) - (i,i,i) language c as
 "_fg_revmask"
determ fg_save(integer,integer,integer,integer) - (i,i,i,i)
 language c as "_fg_save"
determ fg_savew(real,real,real,real) - (i,i,i,i) language c as
                              "_fg_savew"
determ fg_scroll(integer,integer,integer,integer,integer,integer)
                              - (i,i,i,i,i,i) language c as "_fg_scroll"
determ fg_setangle(real) - (i) language c as "_fg_setangle"
 determ fg_setattr(integer,integer,integer) - (i,i,i) language c
                              as "_fg_setattr"
determ fg_setcaps(integer) - (i) language c as "_fg_setcaps"
 determ fg_setclip(integer,integer,integer,integer) - (i,i,i,i)
                              language c as "_fg_setclip" determ
 fg_setclipw(real,real,real,real) - (i,i,i,i) language c as
                              "_fg_setclipw"
determ fg_setcolor(integer) - (i) language c as "_fg_setcolor"
 determ fg_setdacs(integer,integer,integerlist) - (i,i,i)
                               language c as "_fg_setdacs_0"
determ fg_setfunc(integer) - (i) language c as "_fg_setfunc"
determ fg_sethpage(integer) - (i) language c as "_fg_sethpage"
determ fg_setmode(integer) - (i) language c as "_fg_setmode"
determ fg_setnum(integer) - (i) language c as "_fg_setnum"
determ fg_setpage(integer) - (i) language c as "_fg_setpage"
determ fg_setratio(real) - (i) language c as "_fg_setratio"
determ fg_setrgb(integer,integer,integer,integer) - (i,i,i,i)
                              language c as "_fg_setrgb"
determ fg_setsize(integer) - (i) language c as "_fg_setsize"
determ fg_setsizew(real) - (i) language c as "_fg_setsizew"
determ fg_setvpage(integer) - (i) language c as "_fg_setvpage"
determ fg_setworld(real,real,real,real) - (i,i,i,i) language c as
                               "_fg_setworld"
determ fg_sound(integer,integer) - (i,i) language c as
                              "_fg_sound"
determ fg_stall(integer) - (i) language c as "_fg_stall"
determ fg_tcmask(integer) - (i) language c as "_fg_tcmask"
determ fg_tcxfer(integer,integer,integer,integer,integer,integer,
integer,integer) - (i,i,i,i,i,i,i,i) language c as "_fg_tcxfer"
determ fg_text(string,integer) - (i,i) language c as "_fg_text"
determ fg_transfer(integer,integer,integer,integer,integer,
                              integer,integer,integer) - (i,i,i,i,i,i,i,i)
                               language c as                                "_fg_transfer"
determ fg_version(integer,integer) - (o,o) language c as
                              "_fg_version"
determ fg_voice(integer,integer,integer,integer) - (i,i,i,i)
                              language c as "_fg_voice"
determ fg_waitfor(integer) - (i) language c as "_fg_waitfor"
 determ fg_waitkey - language c as "_fg_waitkey"
determ fg_where(integer,integer) - (o,o) language c as
                              "_fg_where"
```

/* fastgraf.rps */

/*--
I compile all my modules in the Borland Integrated Developers Environment, and have it compile the '.obj' in a special
directory for the pupurpse c:\fastgraf\store. The following is the response file I used to call tlib.exe do bundle all
functions in the Prolog-'C' interface into the fastgraph library. The files have rather funny names, because the files cannot have the same name as the 'C' files that Mr. Gruber used to build the oridinal 'fgl.lib' With just a few exeptions I have gave the the 'C' their most obvious names as I wrote the code; the file containing the function to find the best available video mode I called bestmode.c, and the file contain the code to set a pixel to specific colour I called setpixel.c
Then, to ensure the names of the files were unique (different from Gruber's) I took the first six letters of the obvious name, and appended '_p' to them, and used this as the filename (followed by the extension '.c' of course. The names are awkward, admittedly, but this seemed the best way to go. If you use a different directory structure, you'll have to make the appropriate changes--but at least it's simple to make them.
--*/

+ c:\fastgraf\store\arryli_p + c:\fastgraf\store\alloca_p & + c:\fastgraf\store\allocc_p + c:\fastgraf
\store\alloce_p & + c:\fastgraf\store\allocx_p + c:\fastgraf\store\automo_p & + c:\fastgraf\store
\bestmo_p + c:\fastgraf\store\button_p & + c:\fastgraf\store\capslo_p + c:\fastgraf\store
\dsppcx_p & + c:\fastgraf\store\dspprf_p + c:\fastgraf\store\egache_p & + c:\fastgraf\store
\ems_pro + c:\fastgraf\store\freepa_p & + c:\fastgraf\store\getadd_p + c:\fastgraf\store\getclo_p
& + c:\fastgraf\store\getcol_p + c:\fastgraf\store\getdac_p & + c:\fastgraf\store\gethpg_p + c:
\fastgraf\store\getind_p & + c:\fastgraf\store\getkey_p + c:\fastgraf\store\getmxx_p & + c:
\fastgraf\store\getmxy_p + c:\fastgraf\store\getmod_p & + c:\fastgraf\store\getpag_p + c:\fastgraf
\store\getpix_p & + c:\fastgraf\store\getvpg_p + c:\fastgraf\store\getxjo_p & + c:\fastgraf\store
\getyjo_p + c:\fastgraf\store\getxpo_p & + c:\fastgraf\store\getypo_p + c:\fastgraf\store\initjo_p &
+ c:\fastgraf\store\maprgb_p + c:\fastgraf\store\makpcx_p & + c:\fastgraf\store\measur_p + c:
\fastgraf\store\memava_p & + c:\fastgraf\store\mousei_p + c:\fastgraf\store\numloc_p & + c:
\fastgraf\store\palett_p + c:\fastgraf\store\pcx_cols &
c:\fastgraf\store\pcx_size &
c:\fastgraf\store\playin_p + c:\fastgraf\store\polygo_p & + c:\fastgraf\store\scrloc_p + c:\fastgraf
\store\setdac_p & + c:\fastgraf\store\sounds_p + c:\fastgraf\store\testmo_p & + c:\fastgraf\store
\xalpha_p + c:\fastgraf\store\xconve_p & + c:\fastgraf\store\yalpha_p + c:\fastgraf\store
\yconve_p & + c:\fastgraf\store\xms.pro

/* xfast.rps */

+c:\fastgraf\store\polygw_p + c:\fastgraf\store\swleng_p & +c:\fastgraf\store\xscree_p + c:
\fastgraf\store\xworld_p &
+c:\fastgraf\store\yscree_p + c:\fastgraf\store\yworld_p

..\..\prolog
+\borlandc\lib\emu+\borlandc\lib\mathl+\borlandc\lib\cl+profast+xprofast

```
/* fg_link.bat */
optlinks /CO /LI /DET tcfirst c:\borlandc\lib\c0l tcinit cmain
                        %1 %2 %3 %4
%1.sym,%1.exe,%1.map,@fastgraf.inp
```