procedure and function statements don't have *public*, FORWARD or *external* statements following them. Nor do they have <block>s following them. They are assumed to be *external*, since they are found in the *interface* Block.

VAR statements also cannot have *public*, *external* or STATIC attributes associated with them since they are assumed to be *external*.

## 6.7 Use Of INTERFACE Block

In general, you will create an *interface* block for each *module* that you create. The *module* will contain all the *public* definitions and will be compiled to create an object module that contains those procedures, functions and variables. The *interface* module will exist only in the form of Pascal source code and contain the *external* (by default) definitions that are then used in all the other *modules* (or the *program*) that reference this *module*.

In our graphics example, we might have the following *module*:

```
MODULE HiResolution;

CONST  ScreenSize = $1800;

TYPE   ScreenByte = -128..127; (* 1 Byte Integer *)
       Screen = ARRAY[1..ScreenSize] OF ScreenByte;
       GraphTypes = (GTalpa, GTsemi4, GTsemi6, ...);

VAR PUBLIC
       GraphMode : GraphTypes;

PROCEDURE MapScreen (VAR ScreenVar : Screen);
PUBLIC;
  .
  .    (* procedure block *)
  .

PROCEDURE ClearScreen (VAR ScreenVar : Screen);
PUBLIC;
  .
  .    (* procedure block *)
  .

  .
  .    (* other public and private procedures
  .    and functions required for package *)
  .

END.
```

This *module* contains a number of *public* interfaces including procedures, functions and at least one variable. Another *module* which is responsible for creating pie-charts may reference this *module* as follows:

```
INTERFACE HiResolution;

CONST  ScreenSize = $1800;

TYPE   ScreenByte = -128..127; (* 1 Byte Integer *)
       Screen = ARRAY[1..ScreenSize] OF ScreenByte;
       GraphTypes = (GTalpa, GTsemi4, GTsemi6, ...);

VAR    GraphMode: GraphTypes;

PROCEDURE MapScreen (VAR ScreenVar : Screen);

PROCEDURE ClearScreen (VAR ScreenVar : Screen);

.

END;

.

MODULE PieCharts;

.
.

END.
```

The module *PieCharts* uses the module *HiResolution* and sees its interface to *HiResolution* in terms of the *interface* block. Note that in general, the source code comprising the *interface* block will be in an independent file which is copied at compile time via the compiler *%C* directive.

One final note, the file *PASCALIB/EXT* is actually an *interface* block with a *%N* at the beginning and a *%L* at the end which is automatically copied by **DEFT Pascal** at the beginning of every compilation. You can force it to be listed by including an *L* directive in the directive prompt on the compiler startup screen.

# 7 Assembler Interface

One of the primary advantages to using both **DEFT Pascal** and **DEFT Bench** is the ability to *easily* mix Pascal and assembler language as appropriate in the development of a program. This section provides the information on using variables, procedures and functions from assembler and in turn creating variables, procedures and functions in assembler for use from Pascal, with **DEFT Pascal.**

A pre-requisite required for this section is a familiarity with the Motorola 6809 Assembler Language, and the **DEFT Macro/6809** Assembler. Information on linking object files produced by the **DEFT Pascal** Compiler and the **DEFT Macro/6809** Assembler can be found in the section on the **DEFT Linker.**

## 7.1 Code Generation Strategy

The **DEFT Pascal** compiler is a single-pass, recursive descent compiler which directly produces 6809 object code suitable for linking by **DEFT Linker.** In order to produce this object code, a *code generation strategy* is required so that the state of the machine can be predicted from statement to statement. This strategy defines how code, data and stack memory areas are organized as well as how the 6809 registers are used. In addition, the actual memory organization of all the various Pascal types should be understood.

### Variable Sizes and the Stack

As can be guessed by the ordinal and pointer *types* available with **DEFT Pascal,** the language is 16 bit oriented. To a large extent this is due to the registers and functions available on the 6809. By keeping to a 16 bit organization, the resulting compiler is both smaller and more efficient.

In general, all instructions generated by the compiler are oriented around the program stack. As factors are encountered in an expression, they are pushed on the stack. Operators then operate on the top of the stack or combine the top two elements of the stack to form a result which is left on the top of the stack.

The number of bytes of data pushed on the stack depends on the *type* of the expression. The following table shows the number of bytes for each *type:*

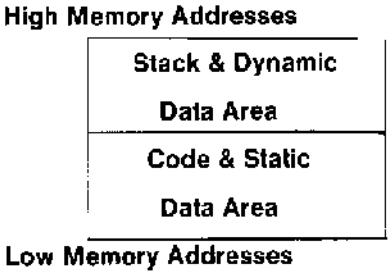
| | |
|---|---|
| **ordinal type** | **2 bytes** |
| **pointer type** | **2 bytes** |
| **real type** | **7 bytes*** |
| **set type** | **32 bytes** |
| **file type** | **286 bytes + type size** |
| **string type** | **string size + 1 bytes** |
| **array and record types** | **sum of components** |

Although real types have a size of 6 bytes, when a real type is pushed on the stack, an additional byte is added in order to limit loss of precision during arithmetic operations. The symbol table printed at the end of each block shows the size of all the variables and types defined within that block.

Any time parameters are passed to a procedure or function, they are first pushed onto the stack. Values returned by functions are left on the stack when the function returns.

## Memory Organization

The general memory organization of a Pascal program is shown in the following diagram:




As can be seen from the above diagram, the code and static data items are allocated in low memory and the stack with its associated dynamic data items are allocated in high memory.

The code and static data items are interspersed in the order in which they were encountered by the compiler. The code and static data area is built from low addresses to high addresses by the compiler. The resulting area is what is linked by **DEFT Linker** and eventually loaded via the *LOADM* command. Because **DEFT Linker** essentially handles all the code and static data linkage, the actual organization of memory is of little concern to the programmer.

The stack and dynamic data area is organized by the compiler but not actually allocated until execution of the resulting program. As a result the actual memory addresses cannot be predicted. The organization of this stack area is the key to interfacing Pascal and Assembler.

## Register Usage

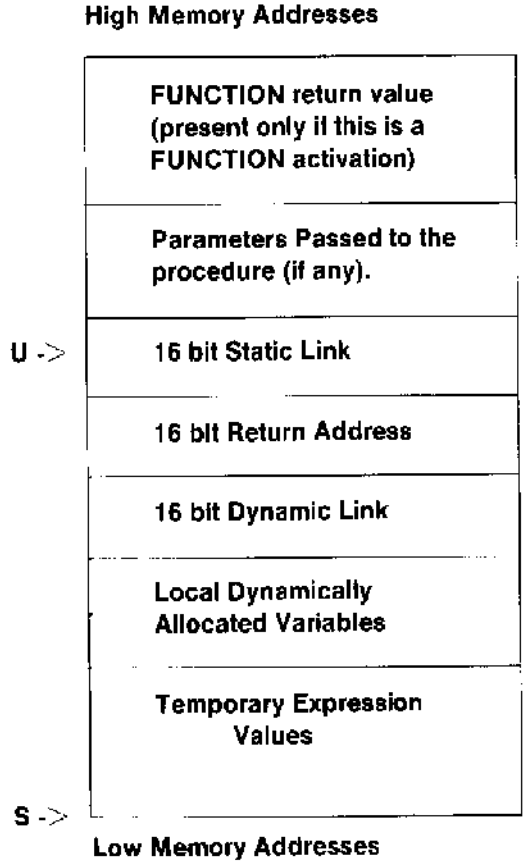
The use of the registers is oriented around the stack. The following lists the 6809 registers and summarizes their use:

- The *S register* is the program stack register. It always points to the youngest element on the stack. This stack always grows or shrinks by the size of the *type* begin pushed or popped. Elements are added by decrementing the S register and are removed by incrementing the S register.
- The *D register* is the primary accumlator, and so is considered to be the *top of stack* for most operations. This is done by placing data in the *D* register before actually pushing it on the stack. Data is popped from the stack into the *D* register. By considering the *D* register to be the *top of stack*, operating on the top stack element is easy with the 6809 instruction set.
- The *U register* is the *frame pointer*, which identifies that group of data on the stack which is associated with the most recent procedure activation. See the section on *Procedure Frame Structure* for a complete description.
- The *X register* is used as a secondary *frame pointer*, when traversing the static frame links in order to access an identifier which is *global* to a procedure. See the section on *Procedure Frame Structure* for a complete description. It is also used for array indexing and variable addressing.
- The *Y register* is used for temporary storage, loop counting and compare operations.

On return from a procedure or function, only the *U*, *S* and *DP* registers will be preserved. All other registers may have been modified.

## 7.2 Procedure Frame Structure

A *frame* is a contiguous portion of the stack that contains all the dynamic information relating to a specific procedure activation. Anytime a procedure or function is invoked, a frame is pushed onto the stack. The structure of a frame is:




The *base* of the frame is the *static link*. The U register always contains the base address of the most recently active frame (last one pushed on the stack). The following notes apply to the individual fields of the frame:

1. The function return value is only present on a function activation and can be considered to be the "zeroth" parameter.

2. The parameters are pushed on the stack in the order in which they occur in the <parameter list>. That is, the first parameter has the highest memory address and the last parameter has the lowest memory address. Each occupies the amount of memory specified in the section on *Variable Memory Requirements.*

3. The *static link* contains the base address of the most recent frame activation for the immediately enclosing procedure. This address is used when referencing variables which are global to the current procedure.

4. The 16 bit return address is the last element of the frame that is created by the *calling procedure* with a JSR or BSR instruction. The *called procedure* creates the remainder of the frame before executing its first statement.

5. The 16 bit *dynamic link* is the base address of the *calling procedure's* frame. It is placed on the stack by the *called procedure* via a PSHS U instruction. The U register is then immediately reset to the current frame's base address via a LEAU 4,S instruction.

6. The local, dynamically allocated variables are then allocated via an LEAS -n,s instruction which *only allocates and does not initialize.*

7. As <executable statement>s are executed additional stack space is used for temporary, intermediate expression values.

Returning from a procedure is easily accomplished with the following two instructions: LEAS -4,U and PULS U,PC. The *calling procedure* is then responsible for removing the parameters from the stack and using the function return value (if there is one).

The reason for having separate static and dynamic links is to provide for the ability to handle recursive procedure (or function) activation. The static link provides execution time identifier scoping, regardless of the number of times the current procedure has activated itself. The dynamic link provides the ability to return to the frame that activated the current procedure (or function).

As can be seen, as long as the assembly language program obeys these rules, it can either invoke a Pascal procedure or function or be invoked as if the assembly language procedure or function was written in Pascal.

## 7.3 Linkage

Linkage between Pascal *modules* is implemented via *public* and *external* attributes and statements as described in previous sections. Linkage to assembly language modules is exactly the same.

You can declare your own Pascal callable routine as *public* in your assembly language program so that it is visible to **DEFT Linker**. You then use the same name to declare the corresponding *external* procedure or function in the Pascal *module(s)* from which it is to be called. The same is true of shared, static variables which would be declared as *public* in your assembly language modules and *external* in the appropriate Pascal *module*.

Alternatively, you can create a Pascal *interface* that corresponds to your assembly language module in order to provide a more formal interface. All Pascal *modules* that reference any of your assembly language procedures, functions or variables would then %C the *interface module* to the beginning of their code. Language identifiers that are declared *external* in Pascal must be declared as *public* in your assembly language program.

Any Pascal procedures, functions or variables you wish to access from assembly language, must be declared as *public* in the corresponding Pascal *module* or *program*. The identifiers are then declared as *external* (via the EXT directive) in your assembly language program.

## 7.4 Initialization

All programs produced by **DEFT Linker** have a first instruction. For Pascal programs produced with the **DEFT Pascal** Compiler, this is in the runtime support module named PASBOOT. This is the module that determines the amount of memory in your system, sets the stack pointer appropriately, sets up all interrupt vectors for the device drivers, setups the initial frame on the stack and then calls the main pascal program.

## 7.5 PIC and ROM

The code produced by the **DEFT Pascal** Compiler is generally position independent and non-selfmodifying (can be placed in Read Only Memory-ROM). There are certain conditions under which this is not true:

1. Any presence of *static* or *public* variables within a Pascal program will result in a module that is self-modifying.

2. Any procedure, function or variable that is declared as *external* in a Pascal program, and whose actual address is an absolute memory location, will result in a module that is not position independent. Absolute memory access can be accomplished in **DEFT Pascal** via the BYTE, WORD and CALL language elements so that the resulting module *will* be position-independent.