If you have any comments on this Guide, please complete and return the form at the back of the Guide to the address given there. All other correspondence should be addressed to:

Customer Support and Services
Acorn Computers Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN

Published July 1989
Published by Acorn Computers Technical Publications Department
Part Number 0483,764
Issue 1

Contents

About this manual

Readership of this manual

This manual describes those areas of the RISC iX operating system that are of interest to the programmer writing or porting software for the RISC iX system.

The manual contains information not readily available off-the-shelf. It consists of separate chapters, each chapter dealing with a part of the system and existing as a manual in its own right.

The manual is split into two volumes. This volume contains programming information about languages and the kernel. Volume 2 contains programming information about the Network File System (NFS).

Chapter summaries for volume 1

ARM assembler (as) – describes the UNIX assembler for the Acorn RISC Machine (ARM).

RISC iX ARM procedure call standard – discusses the compiler implementation on the ARM.

C compiler (cc) – outlines the main features of the RISC iX C compiler for the ARM.

FORTRAN compiler (F77) – discusses the implementation of RISC iX FORTRAN.

Writing device drivers – explains how to add software drivers for new devices to the RISC iX kernel.

Shared libraries – describes the RISC iX shared library scheme.

Squeezed image files – describes the disc compression techniques used by RISC iX to make better use of disc space.

For general UNIX programming information refer to the Berkeley 4.3 UNIX documentation set. It contains the following manuals:

- *User's Reference Manual* (URM)
- *User's Supplementary Documentation* (USD)
- *Programmer's Reference Manual* (PRM)
- *Programmer's Supplementary Documents*, Volume 1 (PS1)
- *Programmer's Supplementary Documents*, Volume 2 (PS2)
- *System Manager's Manual* (SMM)

These are available from the EUUG, Owles Hall, Buntingford, Herts, SG9 9PL.

Chapter 1 – ARM assembler (as)

ARM assembler (as)

1.1 Introduction

This chapter describes as, the RISC iX Assembler for the Acorn RISC Machine (ARM), known as the *RISC iX ARM assembler* (as).

as differs from AASM (the Archimedes assembler) and from OBJASM in that it is intended for use in conjunction with the RISC iX C compiler. as accepts standard UNIX as directives, but ARM assembler language.

A detailed list of the differences between as and AASM is given later in this chapter. The differences are reasonably substantial, consisting of lexical, syntax and semantic differences.

1.2 Structure of as output

Division of as output

The output of as consists of the following:

- A file header, which identifies the output as linker input, and defines the size and position of the other sections.
- Text, data and BSS sections (BSS is not present in output – only in the header) into which the code and data generated by the input is placed.
- Relocation records to denote where the link editor, known as the *loader* under RISC iX, is to adjust regions of the text and data sections once the value of undefined symbols, or offset of the various areas are known.
- A symbol table of defined symbols (local symbols are included) and undefined symbols not produced by as.
- A string table which gives the characters in global names referred to within the symbol table.

In the case of the a.out format, the relocation records follow the symbol table.

The format of these sections is not defined here; please refer to the manual page describing a.out.

Text, data and BSS

The two sections of output corresponding most closely to the input language are known as text and data, and follow each other in the output file. bss does not occur in the output, only in the header file. as directives .text, .data and .bss are used to signify that the following instructions or directives apply to and are to be assembled into the respective section of the output file.

The user must *not* assume that these sections follow one another in memory in the final runnable process image, despite the fact that the addresses on the listing output would appear to suggest this. The final order and position of sections are under the control of the loader, which by default puts the text sections of all object modules together, followed by all the data sections, and finally all the BSS sections together to yield three overall text, data and BSS sections (two in output, three in memory).

Text

The text section of the output often becomes read only, and in some cases execute only in the final process image. It usually consists almost entirely of instructions, although in some cases data intended to be read only may be assembled there. Code constants, and the `.pool` directive are supported within instructions intended for this section.

If space is reserved but not defined within the text section, it is zero filled.

Data

The data section of the output always has read write access in the final process image. It usually consists entirely of data definitions, although instructions which do not involve code constants may be assembled.

As with the text section, space reserved but not defined within the data section is zero filled.

BSS

The BSS section consists of uninitialised data. Only an indication of its size is given in the output file, and the entire section is zero filled by the RISC iX kernel when the process image is run.

Space may only be reserved (using the `.space` directives) within the BSS section. Any form of opcode or data defining directive, even one which would assemble a zero, is regarded as an error by `as`. Note, `.comm` does not reserve contiguous space in BSS.

`.comm` does *not* reserve space in any section in any output; it specifies the maximum 'size' of an external symbol.

An example in C

The division of `as` output into text, data and bss reflects the output of the C compiler, which relies upon `as` to perform a sorting operation.

Thus if the following C program in a file named `foo.c` is compiled:

```
char        data[ ] = "This is in data";
char        bss[100];
fn()
{
            another_fn();
}
```

The following `as` output is obtained:

```
            .file           "foo.c"
            .data
            .global         data
data:
            .ascii          "This is in data\x00"
            .comm           bss,100
            .text
            .global         fn
fn:
            stmfd           sp!,{fp,rl}
            mov             fp,sp
            sub             sp,fp,#4
            bl              another_fn
            mov             sp,fp
            ldmfd           sp!,{fp,pc}^
            .pool
```

1.3 Assembler language

The language accepted by as consists of two types of statement;

- Instructions.
- Assembler directives.

Assembler directives are further subdivided into:

- Instruction definition, data definition or reservation statements.
- Symbol definition and debugging statements.
- Listing control statements.

Each statement may appear on a line by itself, or several statements may appear on one line separated by semicolons. Comments are introduced by the at sign (@, not within a string) or by lines beginning with a #; these cause the rest of the line to be ignored. Blank lines are ignored, as are spaces and tabs (except where these separate identifiers) within the input.

The following is an example of valid as input.

```
@This is a comment
label:      mov             r1,r2           @ another comment
            adds            r2,r3,r4
            mov             r0,#'@'         @ this is a comment
            .nolist
            .data
            .long           1,2,label-3
            .text
            b               label
```

Identical output would be obtained from:

```
label:mov r1,r2;adds r2,r3,r4
mov r0,#'@';.nolist;.data
.long 1,2,label-3;.text;b label
```

In the interests of readability, however, heavy use of the placing of multiple statements on one line is not recommended. The C compiler only places one statement per line.

as syntax

In this section the syntax of data reservation statements and opcodes is given. Listing control statements are described individually later.

The following general format applies:

[label:] instruction [arguments]

Labels are optional, and consist of one or more identifiers or local symbols (defined below), each terminated by a colon (:). Note that this is a key difference between as and AASM, which distinguishes labels by the position on the line.

Many of the instructions require alignment onto a boundary which is a multiple of four. If such an instruction is labelled, then any necessary padding out to such a boundary is done before the label is assigned.

One or more spaces or tabs must be inserted between the instruction and its arguments to separate them.

Identifiers

Identifiers consist of any contiguous block of upper or lower case letters, digits, the full stop character or the underscore character, but not starting with a digit. The following are examples of valid identifiers:

```
r3              .L12
a_long_name_34_b
SWI             stmfd
foo.bar         pc
```

Note in particular that identifiers may collide with register or opcode names, but that this is strongly discouraged if readability is to be maintained.

Local symbols

A label, instead of being an identifier, may be a decimal number in the range 0 to 255. The statement thus labelled may be referred to in the input in the immediate vicinity of the label by means of the notation *number*`f` and *number*`b` to denote the next following and the last preceding statement labelled with *number*.

Such labels do not appear either in the symbol table or the cross reference listing, saving identifiers for important points in the code rather than for small loops or jumps, for example:

```
@ Count bits in r0, result in r2

                mov             r2,#0
                cmp             r0,#0
1:              addne           r2,r2,#1
                subne           r3,r0,#1
                andnes          r0,r0,r3
                bne             1b
```

Note that the local symbol nearest in the source text to the *number*`f` or *number*`b` applies, not the nearest such symbol in the same section. Thus in:

```
                ldr             r0,#0f
                str             r0,[sp,#-4]!
                bl              printf
                add             sp,sp,#4
                .data
0:              .asciz          "Hello world\n"
                .text
0:              ldmfd           sp!,{fp,pc}^
```

The `#0f` applies to the `.asciz` statement, not to the text statement two lines further down (even though the code constant corresponding to the `#0f` is assembled later, possibly in the vicinity of other local symbols `0`).

A backward reference to a local symbol not previously defined, or a forward reference to one not subsequently defined, is an error.

Current location counter

The symbol `.` (full stop) represents the current location counter, which is calculated before the location counter is advanced following assembly of the current expression.

Register and shift symbols

Certain identifiers are predefined to be register and shift symbols, and as such have meaning in CPU instructions which take registers and shift types as arguments.

The following four classes of symbol are available:

- register names
- floating point register names
- shift names (apart from rotate right with extend)
- shift names for rotate right with extend (treated with the other shift types).

Synonyms for any or all of these symbols may be defined using the `.req` directive.

Register names

The following are predefined:

Name	Number
r0-r15	0-15
a1-a4	0-3
v1-v6	4-9
sl	10
fp	11
ip	12
sp	13
lr	14
pc	15

Floating point register names

The names `f0` to `f7` are predefined to denote floating point registers. These names may also be given synonyms via the `.req` directive.

Shift names

The following names are predefined:

Name	Meaning
asl	arithmetic shift left
lsl	logical shift left
asr	arithmetic shift right
lsr	logical shift right
ror	rotate right
rrx	rotate right with extend

In fact arithmetic and logical left shift are synonyms (the ARM does not generate an overflow if the sign bit is changed during an arithmetic shift).

Note that these names can also be given synonyms via the `.req` directive. For example:

```
okey_cokey      req             rrx
twirl.req       ror
                mov             r0,r1,okey_cokey
                add             r7,r8,r9,twirl #10
```

Constants

A constant may be represented by any of the following:

- A decimal number, such as `27`, `0`, `1234` etc.
- An octal number, which consists of a string of digits 0 to 7 preceded by an initial 0 (zero), such as `0177`.
- A hexadecimal number, consisting of digits and the letters `a` to `f` or `A` to `F` preceded by an initial `0x` (zero x) or `0X` (zero X). (The cases of the letters are insignificant.)
- A number of arbitrary base, written in the form *base_number* where *base* is decimal and is 2 or more and *number* consists of digits and alphabetic characters whose case is ignored where `A` represents 10, and so on up to `Z` which represents 36. For example 255 in base 32 arithmetic may be written `32_7U`.

A character is not permitted whose value exceeds the base. There is no way of representing the character 37 upward.

- An ASCII character, which should be enclosed in single quotes, for example 'a' or '@'. To represent a single quote or a backslash, it should be preceded by a backslash thus '\'' and '\\'.
- Control characters may be represented by up to three octal digits preceded by a backslash thus, `\007`, or by two hexadecimal digits preceded by a backslash and x, thus `\x0C`.

  A special notation is available for the control characters tab, newline, formfeed, backspace, carriage return and vertical tab, which may be represented respectively by `\t`, `\n`, `\f`, `\b`, `\r` and `\v`.
- A register list. This is defined below.

Register list constants

Although primarily intended for use in conjunction with `stm` and `ldm` instructions, the construct `{register list}` represents a 16 bit constant.

A register list may be a list of register names (not floating point registers) separated by commas, as in:

```
{r1,r2,r3,r4,r5}
```

Alternatively a sequence of registers may be written by writing the first and last separated by a hyphen as in:

```
{r1-r5}
```

by which the previous example may be replaced.

It is an error to mention a register twice in a register list, either explicitly or as part of a sequence of registers.

Expressions

An expression consists of identifiers and constants, together with the following operators in descending order of binding power.

Operator	Meaning
`* / %`	Multiply, Divide, Remainder
`+ -`	(Binary) Plus or Minus
`<< >>`	Left, Right arithmetic shift
`&`	Bitwise AND
`^`	Exclusive OR
`|`	OR

The binding of operators may be overridden in the usual way by means of parentheses, as in `(x+9)*17`.

Relocatable and absolute expressions

An expression which contains a reference to a symbol or symbols not defined within the source input (an external symbol), or to a symbol labelling a text, data or bss location defined within the source, is regarded as relocatable. An expression may be relocated more than once, or by a negative amount if a relocatable symbol is subtracted.

The difference of two symbols from the same section is not relocatable. In fact `as` counts the number of text, data and bss relocations in the expression, and if all are zero, and there are no external symbols, the expression is regarded as an absolute expression (one which requires no relocation). Thus:

```
Text1*3 - Data1*2 + Data2 + Data3 - Text2 - Text3*2
```

would be regarded as absolute. Any external symbol or symbols makes the expression relocatable, even if a difference is taken, and this would certainly be an error.

Relocatable expressions are only permitted as:

- branch destinations
- 32 bit words (`.long` or code constants)
- the argument in `swi` instructions.

Immediate constants

Instructions which operate between registers may take immediate constants, which consist of 8 bit quantities rotated right by a constant amount, which must be a multiple of two. Such constants may be represented in two ways:

- The constant may be represented as an absolute expression which evaluates to a valid 8 bit quantity rotated right by a multiple of two, such as:

```
0xff
0x1e00
93
72 << 12
```

- The constant may be represented as an absolute expression which evaluates to an 8 bit quantity, followed by a comma and another absolute expression which yields an even number in the range 2 to 30, for example:

```
17,4
0xff,0x10
```

Floating point immediate constants

Some floating point instructions may take one of eight immediate constants, which may only be written as constants and not as expressions, and which are encoded in three bits as follows:

Constant	Coding
0.0	000
1.0	001
2.0	010
3.0	011
4.0	100
5.0	101
0.5	110
10.0	111

Code constants

The following instructions may take code constants introduced by a # character, provided that they are assembled in the text section.

Instruction	Expression	Data Type	Description
`ldrb`	Yes	Byte	Load byte
`ldr`	Yes	Word	Load
`ldfs`	No	Float	Load single floating point
`ldfd`	No	Double	Load double floating point
`ldfe`	No	Extended	Load extended floating point
`ldfp`	No	Packed	Load packed decimal floating point

The type of the constant is everywhere inferred from the instruction, and is subsequently inserted by means of the `.pool` directive.

Global names

An identifier is regarded as global if it is mentioned in a `.global` directive, or if it is used but not defined anywhere in the source file.

An identifier which is mentioned in a `.global` directive and defined within the source file is an external symbol defined within that source file, otherwise it is an undefined external symbol.

The only distinction between an undefined external symbol which does or which does not appear in a `.global` directive is that the former is appears in the cross-reference listing, if one is requested.

1.4 CPU Instruction set

The following instructions are supported:

- Branch instructions
- Data operations between registers. These are further subdivided into:
  - 4 comparison instructions
  - 2 data movement instructions
  - 10 arithmetic and logical instructions
  - 2 multiply instructions
  - 1 pseudo-instruction for addressing.
- Single register load and store instructions
- Load and store multiple instructions
- Software interrupt instruction
- Floating point instructions. These are further subdivided into:
  - Data transfer instructions
  - Double operand arithmetic instructions
  - Triple operand arithmetic instructions
  - Register transfer instructions
  - Floating–point comparison instructions.

Conditions

All ARM instructions take a condition, which is encoded in the most significant four bits of the instruction. `as` assumes the *always* condition as a default, however the following two letter codes may be used to specify different conditions. The two letters are always inserted into the instruction mnemonic as the fourth and fifth characters, except in the cases of branch instructions (which are one or two characters), to whose mnemonics the letters are appended.

Code	Bits	PSW bits	Explanation
eq	0000	Z=1	Equal
ne	0001	Z=0	Not equal
cs	0010	C=1	Carry set
hs	0010	C=1	Unsigned higher or same
cc	0011	C=0	Carry clear
lo	0011	C=0	Unsigned lower
mi	0100	N=1	Minus
pl	0101	N=0	Plus
vs	0110	V=1	Overflow set
vc	0111	V=0	Overflow clear
hi	1000	C=1 and Z=0	Unsigned higher
ls	1001	C=0 or Z=1	Unsigned lower or same
ge	1010	N=V	Greater or equal
lt	1011	N≠V	Less than
gt	1100	N=V and Z=0	Greater than
le	1101	N≠V or Z=1	Less than or equal
al	1110	any	Always (default)
nv	1111	any	Never (i.e. no-op)

The following are examples of how the condition is inserted into instruction mnemonics to yield a conditional mnemonic.

Instruction	With condition	Becomes
str	eq	streq
ldrb	ne	ldrneb
mov	ge	movge
movs	vs	movvss
ldmfd	cc	ldmccfd
b	ne	bne
bl	eq	bleq
swi	ne	swine
stfe	ge	stfgee

In the descriptions of instructions in the sections which follow, the possibility of inserting condition letters is everywhere assumed.

Instruction alignment

Instructions must be aligned on a word boundary, and if labelled, the label applies to the aligned address.

Branch instructions

The syntax of branch instructions is as follows:

```
b       expression
bl      expression
```

The *expression* is turned into a 24 bit signed word offset from the branch instruction. Where the destination proves to be within the same section of the same source file, no relocation is required. Other cases require one or more relocation records to be generated.

Where the destination is to an external symbol (plus or minus a constant), an *absolute branch relocation* is generated. This causes the address of the external symbol, minus the offset of the branch instruction to be added to the constant.

Where the destination is within the same source file, but to a different section, two *relative branch relocations* are generated, one to subtract the base address of the current section, and another to add that of the destination section.

The following are examples of valid branch instructions:

```
b           x
bl          func
beq         10b
b           external+20
b           .                   @ Loop
```

It is an error for the offset not to be a multiple of four. This applies even if the offset consists of an external symbol whose value added to the constant part of a destination, as in `address+1`, would come to a multiple of four.

Eight is subtracted by `as` from the offset given in the instruction, as required by the processor (the program counter being eight bytes in advance). The usual effect of an unresolved external symbol with no offset being assembled is that a *branch to self* instruction is generated.

Data operations between registers

These instructions (with the exceptions of the multiply and pseudo-operation `adr`) take two or three operands, of which all but the last are registers, and the last of which may take the forms of:

- An immediate constant, in one of its two forms, and preceded by a # (sharp, hash or pound sign), for example as in:

```
mov         r0,#255
sub         r1,r2,#47
mvn         r3,#17,4
teq         r7,#4
```

- A register, as in:

```
mov         r0,r9
sub         r1,r2,r10
mvn         r3,r12
teq         r3,r1
```

- A register with a constant shift, as in:

```
mov         r0,r9,lsl #4
sub         r1,r2,r4,rrx
mvn         r3,r12,ror #const/2
teq         r3,r4,lsr #2
```

- A register with a register shift, as in:

```
mov         r0,r9,lsl r3
sub         r1,r2,r8,ror r7
mvn         r3,r7,lsr r10
teq         r3,r4,lsr r9
```

Data movement instructions

The following two instructions move data to a register, specified as the first operand of the instruction.

Instruction	Meaning
mov	Move
mvn	Move negated

The letter `s` may be appended to the mnemonic to denote that the condition bits `N`, `Z` and `C` are to be set by the instruction. Note that `V` is unaffected by these instructions, so that the conditions `vs`, and `vc`, and especially `ge`, `gt`, `le` and `lt` will not always operate as expected.

Examples:

```
mov         1,#0
mov         r2,r3,lsr #2
mvn         r9,#0
movs        r1,r2
mvns        r3,r10
moveq       r1,r2
mvnnes      r3,r4
```

Arithmetic and logical operations

These instructions all take three operands, a destination register, a source register and a source operand.

Instruction	Meaning	Arith/Logical
`adc`	Add with carry	Arithmetic
`add`	Add	Arithmetic
`and`	And	Logical
`bic`	Bit clear	Logical
`eor`	Exclusive or	Logical
`orr`	Inclusive or	Logical
`rsb`	Reverse subtract	Arithmetic
`rsc`	Reverse subtract with carry	Arithmetic
`sbc`	Subtract with carry	Arithmetic
`sub`	Subtract	Arithmetic

The letter `s` may be appended to the mnemonic to denote that the condition bits `N`, `Z` and `C` are to be set by the instruction, and `V` by arithmetic instructions. Note that `V` is unaffected by logical instructions, so that the following conditions `vs`, and `vc`, and especially `ge`, `gt`, `le` and `lt` will not always operate as expected.

Examples:

```
add         r1,r1,#1
orr         r2,r3,r4,lsr #2
rsb         r9,r9,#0
adds        r1,r2,r2
subeqs      r3,r4,r5
```

Comparison operations

These instructions all take two operands, a source register and a source operand.

Instruction	Meaning	Arith/Logical
`cmp`	Compare	Arithmetic
`cmn`	Compare Negative	Arithmetic
`teq`	Test exclusive or	Logical
`tst`	Test bits	Logical

The letter `s` may not be appended to the mnemonic, as it is implied by the instruction. However a `p` may be appended to the mnemonic to cause the destination field of the instruction to be set to 15, which causes the PSR bits to be set when the instruction is executed.

As with other logical instructions `teq` and `tst` do not affect the `V` bit, so that the following conditions `vs`, and `vc`, and especially `ge`, `gt`, `le` and `lt` will not always operate as expected.

Examples:

```
cmp         r1,#0
cmn         r1,r2
tst         r9,#1 << 8
teqp        r15,#0
```

Multiply instructions

There are two forms of multiply instruction, which may take only registers as operands.

```
mul         ra,rb,rc
mla         ra,rb,rc,rd
```

The first form inserts into *ra* the product of *rb* and *rc*. The second form inserts into *ra* the product of *rb* and *rc* plus *rd*. *ra* may not be `r15` or the same as *rb*.

The letter `s` may be appended to the mnemonic to denote that the condition bits `N`, `Z` and `C` are to be set by the instruction. As with other instructions, `V` is unaffected by these instructions.

Examples:

```
mul         r1,r2,r3
mla         r1,r2,r1,r1
muls        r1,r2,r3
mlas        r1,r2,r3,r4
mulges      r9,r8,r9
```

ADR pseudo-op

This is more limited that the corresponding operation in AASM. The syntax is:

`adr` *register,expression*

where *expression* represents a location in the same section as the instruction. The instruction becomes either of:

`add` *register*,pc,*#immediate constant*
`sub` *register*,pc,*#immediate constant*

depending upon the value of the expression.

Single register transfer instructions

Eight mnemonics are provided for these instructions, namely:

Mnemonic	Meaning
`ldr`	Load (word)
`ldrb`	Load byte
`ldrt`	Load from user memory
`ldrbt`	Load byte from user memory
`str`	Store (word)
`strb`	Store byte
`strt`	Store to user memory
`strbt`	Store byte to user memory

Operands provide for the following addressing combinations:

Pre-increment	Plus	Constant offset
Pre-increment with writeback	Minus	Register offset
Post-increment with writeback		Shifted register offset

User memory references are not permitted for preincrement cases.

Preincrement addressing

The operands for pre-increment addresses may take the following forms:

register,expression

The *expression* must evaluate to an offset less 4096 bytes either side of the program counter, which is eight bytes ahead of the instruction. The *register* is loaded or stored from or to the specified offset, using the pc as a base register.

register,#expression

The *expression* is evaluated to make a code constant, subsequently planted by means of a `.pool` directive, and referenced using the pc as a base register. This form is only permitted with load operations.

register, [*base register*,#*expression*]
register, [*base register*,#*expression*] !

The *expression* is evaluated to an absolute constant of in the range –4095 to +4095, and the operand address will be the *base register* plus this amount. In the second case, writeback is specified, and the *base register* will be updated to this value at the completion of the instruction.

register, [*base register*,+*offset register*]
register, [*base register*,– *offset register*]
register, [*base register*,+*offset register*] !
register, [*base register*,– *offset register*] !

The operand address is the *base register* plus or minus the contents of the *offset register*. In the last two cases the *base register* is replaced by this value. The + may be omitted in the first and third cases.

register, [*base register*,+*register*,*shift*]
register, [*base register*,– *register*,*shift*]
register, [*base register*,+*register*,*shift*] !
register, [*base register*,– *register*,*shift*] !

This is similar to the previous cases, except that the offset register is shifted by a constant amount, using the shift operations given in register arithmetic instructions but excluding shifts by the contents of a register. Again the + sign may be omitted.

Examples:

```
ldr     r1,0f
ldr     r10,#extsymb-10
strb    r2,[r3,#12]
ldrb    r4,[r5,#-15]!
ldr     r3,[r8,r9]
ldr     r5,[r0,-r10]!
ldr     r2,[r2,r3,lsr #2]
ldr     r2,[r2,-r3,ror #2]!
```

Post-increment addressing

In this case the base register is always updated by the value of the offset, whether held in a register or not.

The operands for post-increment addresses may take the following forms:

register, [*base register*]

In this case an offset of zero is implied. This is post-increment in order to enable the programmer to reference user-mode addresses.

register, [*base register*,*#expression*]

The *expression* is evaluated to an absolute constant of in the range –4095 to +4095. The operand address is given by the *base register*, which is afterwards incremented by this amount.

register, [*base register*] , +*offset register*
register, [*base register*] , – *offset register*
register, [*base register*] , +*register*,*shift*
register, [*base register*] , – *register*,*shift*

The operand address is given by the *base register*, which is subsequently incremented or decremented by the contents, or the constant-shifted contents of the *offset register*. The + may be omitted.

Examples:

```
ldr         r1,[r7]
strb        r2,[r3],#12
ldrb        r4,[r5],#-15
ldr         r3,[r8],r9
ldr         r5,[r0],-r10
ldr         r2,[r2],r3,lsr #2
```

Load and store multiple instructions

These instructions take the form:

```
ldmia       stmia
ldmib       stmib
ldmda       stmda
ldmdb       stmdb
```

followed by operands of the forms:

base register, *expression*
base register! , *expression*
base register, *expression*^
base register! , *expression*^

The *expression* must evaluate to a 16 bit quantity. Register lists are intended to aid the generation of these.

The ! marker after the base register signifies writeback to the base register. The ^ marker after the expression signifies setting of the PSR bit. It is distinguished from the exclusive or expression operator by its position on the line.

As the instructions are frequently used for maintaining stacks, an alternative rendering of the mnemonics describes whether the stacks are ascending or descending, *full* (pre-increment or pre-decrement) or *empty* (post-increment or post-decrement).

The correspondence of the two kinds of mnemonics is as follows:

Mnemonic	Synonym	Mnemonic	Synonym
`ldmia`	`ldmfd`	`stmia`	`stmea`
`ldmib`	`ldmed`	`stmib`	`stmfa`
`ldmda`	`ldmfa`	`stmda`	`stmed`
`ldmdb`	`ldmea`	`stmdb`	`stmfd`

Examples of these instructions:

```
stmfd       sp!,{r5-r10,fp,rl}
ldmeqfd     sp!,{r5-r10,fp,pc}^
ldmia       r5,{r0-r2}
stmia       r13,0xFFFF
```

The software interrupt instruction

The format of the software interrupt instruction is as follows:

`swi` *expression*

The *expression* may be any value, possibly relocatable, which fits into 24 bits.

Examples:

```
swi     23
swi     0xF 020
swi     val + 3
```

The interpretation of the *expression* is operating system dependent, and especially in the case of RISC iX, additional parameters are passed in registers and/or the condition bits, and are not guaranteed to be constant in specification, although the C library routine interfaces are. Assembler programmers are advised to continue to use these.

Floating-point instructions

There are five classes of floating-point instruction, which are described in the following section.

Data transfer instructions

The format of these instructions is a subset of that the `ldr` and `str` instructions, in that no offset register is permitted, and constant offsets may only be a multiple of four in the range –1020 to +1020.

The instructions are `ldf` and `stf` for load and store, followed by the suffix `s`, `d`, `e` or `p` for single, double, extended or packed floating-point respectively, thus giving:

`ldfs`	load single	`stfs`	store single
`ldfd`	load double	`stfd`	store double
`ldfe`	load extended	`stfe`	store extended
`ldfp`	load packed	`stfp`	store packed

The operands may take the form:

floating register`,#`*constant*
floating register`, [`*base register*`]`
floating register`, [`*base register*`] ,#`*offset*
floating register`, [`*base register*`,#`*offset*`]`
floating register`, [`*base register*`,#`*offset*`] !`

In the first form, a code constant is generated of the appropriate type. This is not permitted with the `stf` instruction.

The second two forms are post-increment forms. The first has an offset of zero. In the second case the base register is incremented by the value of the offset, after use as the operand address.

The final two forms are pre-increment with and without writeback.

Examples:

```
ldfs        f0,#7.23
stfd        f1,[r11]
ldfe        f3,[r12],#28
ldfp        f4,[r10,#32]
stfp        f4,[r11,#20]!
```

Double operand arithmetic instructions

The following instructions are available, each storing to the first argument (a floating register), the result of the operation applied to the second, which is another floating point register or constant.

Mnemonic	Meaning
`mvf`	Move
`mnf`	Move negated
`abs`	Absolute value
`rnd`	Round to integral value
`sqt`	Square root
`log`	Log to base 10
`lgn`	Log to base *e*
`exp`	Exponential
`sin`	Sine
`cos`	Cosine
`tan`	Tangent
`asn`	Arc sine
`acs`	Arc cosine
`atn`	Arc tangent

The mnemonics **must** be followed by one of:

`s`	single precision
`d`	double precision
`e`	extended precision

and **may** be followed by one of:

`p`	round towards $+\infty$
`m`	round towards $-\infty$
`z`	round towards 0.

to denote different rounding from *to the nearest*. The arguments take the form:

floating register destination, *floating register source*
or
floating register destination, *floating register immediate*

Examples are:

```
mvfd        f1,#0
sind        f1,f3
mnfeqe      f4,#0.5
```

Triple operand arithmetic instructions

These are similar to double operand instructions, except that two source registers, or one source register and a constant are provided in addition to the destination register.

The opcodes are:

Mnemonic	Meaning
`adf`	Add
`muf`	Multiply
`suf`	Subtract
`rsf`	Reverse subtract
`dvf`	Divide
`rdf`	Reverse divide
`pow`	Power
`rpw`	Reverse power
`rmf`	Remainder
`fml`	Fast multiply
`fdv`	Fast divide
`frd`	Fast reverse divide
`pol`	Polar angle

As before, the mnemonics must be followed by one of:

`s`	single precision
`d`	double precision
`e`	extended precision

and may be followed by one of:

`p`	round towards + ∝
`m`	round towards – ∝
`z`	round towards 0.

to denote different rounding from *to the nearest.*

The arguments take the form:

f-register destination, f-register source1, f-register source2
or
f-register destination, f-register source1, floating immediate2

Examples are:

```
adfd        f1,f2,#3.0
mufd        f1,f3,f4
powged      f4,f7,#4.0
```

Register transfer instructions

These are a mixture of instructions to transfer between ARM and floating-point registers.

Flt instruction

The mnemonic takes the form `flt`, followed by a size `s`, `d` or `e`, and an optional rounding specification `p`, `m` or `z`.

The arguments consist of a floating register destination and an ARM register source.

Examples:

```
flts        f1,r2
fltdz       f4,r9
```

Fix instruction

The mnemonic takes the form `fix`, optionally followed by the rounding specification `p`, `m` or `z`.

The arguments consist of an ARM register destination and a floating register source.

Examples:

```
fix         r2,f1
fixz        r8,f9
```

Control and status register operations

Four instructions are available, namely:

`rfs`	*register*
`wfs`	*register*
`rfc`	*register*
`wfc`	*register*

These instructions read and write the floating-point status and control registers respectively, to or from and ARM register.

Examples:

```
rfs         r1
wfs         r2
rfc         r3
wfc         r4
```

Floating-point comparison instructions

Four instructions are available, thus

Mnemonic	Meaning
`cmf`	Compare floating
`cnf`	Compare negated floating
`cmfe`	Compare floating with exception
`cnfe`	Compare negated floating with exception

Each instruction takes a floating register first argument and either another floating register or an immediate constant for a second argument.

Examples:

```
cmf         f1,#0.0
cnf         f3,f2
```

1.5 Data directives

Data directive layouts

Ten directives are provided to lay out data in either the text or data sections (apart from the code constant facility within the text section), and two to reserve space.

.byte

The directive

`.byte` *expression*
`.byte` *expression,expression...*

where each *expression* is an absolute 8 bit quantity, assembles one or more bytes into successive locations. The alignment of the locations is not considered.

Examples:

```
                .byte           0xff
0:              .byte           10,17*4,2f-1f,'c'
```

.short

The directive

`.short` *expression*
`.short` *expression,expression...*

where each *expression* is an absolute 16 bit quantity, assembles one or more 16 bit words into successive locations. The location counter is padded if necessary to a even–boundary, and any label applies to this even boundary.

Examples:

```
                .short          0xff02
vec:            .short          93*0x75,('a'<<8) | 1
```

.word or .long

The directive

`.word` *expression*
`.long` *expression,expression...*

where each *expression* is arbitrary, and may be relocatable, assembles one or more 32 bit words into successive locations. The location counter is padded if necessary to a boundary whose address is a multiple of four and any label applies to this padded boundary.

Examples:

```
                .long           0xf 10020
0:              .word           1b,extern,23
```

The two directives are synonyms, and are completely interchangeable.

.float, .double, .extend and .packed

The directives `.float`, `.double`, `.extend` and `.packed` are available. These assemble one or more floating-point constants into successive locations. The location counter is padded if necessary to a boundary whose address is a multiple of four and any label applies to this padded boundary.

Examples:

```
            .float          4.9e-2
            .double         8.2e32
10:         .extend         -1e10,1.0,-5.2
            .packed         14e2,7.9
```

.ascii and .asciz

These directives take the form

```
.ascii      "string"
.asciz      "string"
```

For example:

```
            .ascii          "This is a text string"
msg:        .asciz          "Hello world\n"
```

The ASCII characters are assembled into successive bytes, using the same escape notation as for character constants. In the case of `.asciz`, an additional zero byte is appended to the end of the string, as often used by C to terminate a string.

Space reservation directives

.space

The directive

`.space` *expression*

where *expression* is an absolute constant (with no forward references) greater than zero, reserves the specified number of bytes of space. In the cases of text and data sections, the reserved space is filled out with zeroes, in the case of bss, the location counter is just advanced by the relevant amount.

.comm

The directive

`.comm` *identifier,expression*

where *expression* is an absolute constant (with no forward references) greater than zero, names an external symbol, and identifies it as requiring at least the specified number of bytes of space. The link editor takes the largest such requirement for each symbol in each of the object modules it handles and reserves that amount of space in BSS for the symbol, with the identifier starting at the beginning of the space.

No label may be applied to this directive.

There is a deficiency in the `a.out` object file format which does not allow the size to be zero, and this is why a C array size of zero is not permitted.

.odd .even .align

The directives

```
.odd
.even
.align
```

cause the location counter to be advanced to an odd, even or multiple of 4+*n* boundary respectively, padding with zeroes in text or data sections.

No labels may be applied to these directives.

1.6 Further assembly time directives

Section control directives

The directives

```
.text
.data
.bss
```

signify that the ensuing code is to be assembled or space reserved in the specified sections of the object file. No label may be applied to any of these directives.

.pool

The directive

```
.pool
```

causes outstanding code constants to be inserted. This directive may only be used in the text section. A limited amount of comparison is done on code constants, and only one of several identical constants not involving external symbols will be inserted.

Any number of `.pool` instructions may be inserted as necessary to avoid displacement errors in large sections of code.

A `.pool` directive may not be labelled.

.global or .globl

The directive

`.global` *name*, [*name*....]

(`.globl` being a synonym) cause the specified name or names to be made global, i.e. an external symbol. Whether the symbol is defined or not depends on whether or not it appears as a label or `.equ` symbol elsewhere in the source. Undefined symbols are always treated as global.

.equ

The statement

identifier `.equ` *expression*

where *expression* consists of values defined prior to the line involved, or is a constant, gives a value to the specified identifier. This is useful for giving symbolic names to constant values.

Examples:

```
FBIT          .equ          1 << 26
a             .equ          1 << 27
S_ERMODE      .equ          3
IRQMODE       .equ          2
FIQMODE       .equ          1
MODEBITS      .equ          3
```

.req

The directive

identifier `.req` *name*

where *name* is that of a register, floating-point register, or shift name, gives an alternative name to that register or shift. For example:

```
basereg         .req                r13
stack           .req                r12
index           .req                r0
amount          .req                r1
float_work      .req                f2
rotate_with_carry.req               rrx
left_shift      .req                asl

mov             stack,index,left_shift amount
mvfd            float_work,#2.0
```

Debugging directives

as has a restriction in that forward references are not allowed in these directives, as owing to the one-pass nature of as it is inconvenient not knowing which part of the symbol table to place certain information. This is not a serious restriction as the C compiler always generates constants in these directives.

1.7 Listing control directives

The following directives have no effect on the object file, and no effect at all unless a listing option -l, -L, -r, -s, -n or -x is requested when as is invoked.

None of these directives may be labelled.

.title

The directive

```
.title      "string"
```

where the string should not contain any non-printing characters, inserts the specified string at the top of each page of the listing.

The directive will not be listed if it would appear as the first line of a page (without other statements on the same line separated by colons).

Following the first line of the page, if several .title directives appear on a page, the last one will affect the next page.

.page

The directive

```
.page
```

throws a new page on the output listing. The directive itself is not listed if it appears alone on a line.

Listing controls

Three listing controls are available.

```
.list
```

requests a normal listing. This is turned on by the -l option to as. If a line generates more than four bytes of text or data, the excess beyond four is not displayed on the listing.

```
.listall
```

requests a full listing. This is turned on by the -L option to as. All the bytes of text or data are displayed on the listing using additional lines as necessary, with a + sign. following the line number on the second and subsequent additional lines.

For example the source file,

```
.data
.list
.ascii "a string"
.listall
.ascii "a string"
.text
mov r1,r2;mov r3,r4
.list
mov r1,r2;mov r3,r4
```

is listed as:

```
                          1         .data
                          2         .list
00010 74732061            3         .ascii "a string"
                          4         .listall
00018 74732061            5         .ascii "a string"
0001C 676E6972            5+
                          6         .text
00000 E1A01002            7         mov r1,r2;mov r3,r4
00004 E1A03004            7+
                          8         .list
00008 E1A01002            9         mov r1,r2;mov r3,r4
```

The directive `.nolist` turns off listing until the next `.list` or `.listall` directive. If an error message is generated whilst the listing is disabled, the error message is displayed but with the message (`.nolist`) appended.

1.8 Running as

At the simplest case, `as` is run by just giving the name of the source file, for example:

```
as source.s
as prog
```

In this case the object file created is named `a.out`. Any errors are displayed on the standard error output file.

Object file names

An alternative object file name may be specified with the `-o` option, thus:

```
as -o obj.o source.s
```

causes the object file to be created as `obj.o`.

`as` refuses to create object files with suffixes `.s`, `.c`, `.h`, `.l` or `.y`, which are common names of Assembler, C, C include files, *lex* and *yacc* files respectively.

`as` will also refuse to write the object file to a file of the same name as the input file.

The object file is deleted at the end of the assembly if there are errors.

Listing files

If the options `-l` or `-L` are specified, optionally followed by a page length value (defaulting to 62) as in `-l55` or `-L72`, then listing is enabled. The listing option `.list` is initially selected if `-l` is given, and the option `.listall` selected if `-L` is given. The listing type may subsequently be varied by means of listing directives in the source text.

The listing file name is generated by deleting the last suffix, if any, such as `.o` or `.out` from the object file name, and substituting `.lst`.

For example consider the following:

Object file name	Listing file name
a.out	a.lst
obj.o	obj.lst
a.b.c.ob	a.b.c.lst
out	out.lst

Symbol, relocation and line number listings

The options -s, -r and -n cause symbol table and relcation table information to be appended to the listing file.

These are just textual interpretations of the data in the sections of the object file. The user is referred to the relevant a.out manual page for explanation.

Cross reference

The option -x, optionally followed by a page width specification, eg. -x130, causes a cross reference listing to be appended to the listing file. All identifiers in the source text are listed in alphabetical order.

Note that the addresses given for data and bss symbols are relative to the start of the sections, and not as in the source text listing, where the size of the text section is added to the data addresses, and the sizes of the text and data to the bss addresses.

Example as invocation

The command,

```
as -L72 -x130 -o obj.o source.s
```

invokes as to assemble the file source.s, with output to obj.o, and with a full lising to obj.lst, followed by a cross reference.

The page length is set to 72 lines and the page width to 130 columns.

For a full description of the options available with as, consult the as(1) manual page.

1.9 Differences between as and AASM

The following are the main differences between as and AASM.

- There is no conditional assembly or macro facility available in as. The user is expected to use the macro processor m4, or possibly the C preprocessor.
- Mnemonics are in lower case.
- Labels are terminated with a colon.
- The semantics of local symbols are different.
- The second operand of ldm and stm instructions may be an expression yielding a 16-bit constant, and register lists are regarded as a type of constant.
- The syntax of data layout instructions is different.
- Code constants are represented differently.
- as directives are completely different.
- Comment symbols are different.
- Multiple statements on a line are permitted.
- There is no ORG directive or equivalent, the nearest is the .space directive.
- Expression operators are different, being more like C.
- The adr pseudo-op is more restrictive.

- Floating-point instructions and constants are built-in.
- Register names are predefined.
- Synonyms may be given to shifts.
- The layout of the object file, and related directives are tied to RISC iX.

Chapter 2 – RISC iX ARM Procedure call standard

RISC iX ARM Procedure call standard

2.1 Introduction

This chapter discusses the compiler implementation under RISC iX on the Acorn RISC Machine (ARM).

Assumptions

You should be familiar with the ARM's instruction set [ARM], floating-point instruction set [AFP] and assembler syntax [as] detailed in the previous chapter, before attempting to use this information to implement code generators for the ARM. In order to write a run-time system for a language implementation, additional information specific to the RISC iX operating system will also be necessary.

The main topics described in this chapter are the procedure call and stack disciplines. These disciplines are followed in all Acorn language implementations for the ARM. Because C is the implementation language for RISC iX, the utility the usefulness of a new language implementation will be related to its compatibility with Acorn's implementation of C.

At the end of this chapter are several examples of the usage of the standard, with suggestions for generating effective code for the ARM.

For more information about the ARM instruction set read the `aupcs(7)` manual page.

2.2 Intent of the ARM procedure call standard

The ARM Procedure Call Standard is a set of rules, designed

- To facilitate calls between program fragments complied from different source languages (for example, to make subroutine libraries accessible to all compiled languages).
- To give compilers a chance to optimise procedure call, procedure entry and procedure exit (following the reduced instruction set philosophy of the ARM).

This standard defines the use of registers, the passing of arguments at an external procedure call, and the format of a data structure that can be used by stack backtracing programs to reconstruct a sequence of outstanding calls. It does so in terms of *abstract register names*. The binding of some register names to register numbers and the precise meaning of some aspects of the standard are somewhat dependent on the host operating system and are described in separate sections.

Formally, this standard only defines what happens when an *external procedure call* occurs. Language implementors may choose to use other mechanisms for internal calls and are not *required* to follow the register conventions described in this document except at the instant of an external call or return. However, other, system-specific invariants may have to be maintained if it is required, for example, to deliver reliably an asynchronous interrupt (eg, a SIGINT) or give a stack backtrace upon an abort (eg, when dereferencing an invalid pointer).

Design criteria

The procedure call standard was produced after a great deal of experimentation, measurement and study of other architectures. It is believed to be the best possible compromise between various requirements. The following important factors influenced this design:

- The procedure call must be extremely fast.
- The call sequence must be as compact as possible. (Code density on RISC machines is a well-known problem. In typical compiled code, calls outnumber entries by two to five times.
- Extensible stacks and multiple stacks must be accommodated. The standard permits a stack to be extended in a non-contiguous manner, in *stack chunks*. The size of the stack does not have to be fixed when it is created, avoiding a fixed partition of the available data space between stack and heap. The same mechanism supports multiple stacks for multiple threads of control.
- The standard should encourage the production of re-entrant programs, with writable data separated from code.
- The standard must support variation of the procedure call sequence, other than by conventional return from procedure (for example, in support of C's longjmp, Pascal's goto-out-of-block, Modula-2+'s exceptions, Unix's signals, etc.) and tracing of the stack by debuggers and run-time error handlers. Enough is defined about the stack's structure to ensure that implementations of these are possible (within limits discussed later).

2.3 The procedure call standard

The ARM has 16 visible general registers and eight floating-point registers.

In interrupt modes some general registers are shadowed and not all floating-point operations are available, depending on how the floating-point operations are implemented.

This standard is written in terms of the *register names* defined in this section. The binding of certain register names (the 'call frame registers') to register numbers is discussed separately. We do this so that:

- Diverse needs can be more easily accommodated as can conflicting historical usage of register numbers, yet the underlying structure of the procedure call standard – on which compilers depend critically – remains fixed.
- Run-time support code written in assembly language can be made portable between different register bindings, if it obeys the rules given in the section entitled *Defined Bindings*.

Register names

The register names and fixed bindings are given immediately below.

The four argument registers:

```
a1      RN      0   ; argument 1/integer result
a2      RN      1   ; argument 2
a3      RN      2   ; argument 3
a4      RN      3   ; argument 4
```

The six 'variable' registers:

```
v1      RN      4   ; register variable
v2      RN      5   ; register variable
v3      RN      6   ; register variable
v4      RN      7   ; register variable
v5      RN      8   ; register variable
v6      RN      9   ; register variable
```

The call-frame registers, the bindings of which vary (see the section on register bindings for details):

```
sl      RN      10  ; reserved to Acorn
fp      RN      11  ; frame pointer
ip      RN      12  ; used as temporary workspace
sp      RN      13  ; lower end of current stack frame
```

lp and pc, which are determined by the ARM's hardware:

```
lr      RN      14  ; link address on calls/workspace
pc      RN      15  ; program counter and processor status
```

In the obsolete APCS-A register bindings, sp is bound to r12; in ALL OTHER APCS BINDINGS, sp is bound to r13.

References to 'the stack' denoted by `sp` assume a stack that grows from high memory to low memory, with `sp` pointing at the top (ie lowest addressed) word in the stack.

At the instant of an external procedure call there shall be nothing of value to the caller stored below the current stack pointer, between sp and the (possibly implicit, possibly explicit) stack (chunk) limit. Whether there is a single stack chunk or multiple chunks, an explicit stack limit (in sl) or an implicit stack limit, is determined by the register bindings and conventions of the target operating system.

For any register `r`, the phrase 'in `r`' in the following text refers to the contents of `r`. The phrase 'at `[r]`' or 'at `[r,#n]`' refers to the word pointed at by `r` or `r+n`, in line with the corresponding ARM assembly language notation.

Floating-point registers

The floating-point registers are divided into two sets, analogous to the subsets a1-a4 and v1-v6 of the general registers. Registers f0-f3 need not be preserved by a called procedure; f0 is used as the floating-point result register. IN CERTAIN RESTRICTED CIRCUMSTANCES (noted below), f0-f3 may be used to hold the first four floating-point arguments. Registers f4-f7, the so called *variable* registers, must be preserved by callees.

```
f0      FN      0   ; floating point result (or 1st FP argument)
f1      FN      1   ; floating point scratch register (or 2nd FP arg)
f2      FN      2   ; floating point scratch register (or 3rd FP arg)
f3      FN      3   ; floating point scratch register (or 4th FP arg)
f4      FN      4   ; floating point preserved register
f5      FN      5   ; floating point preserved register
f6      FN      6   ; floating point preserved register
f7      FN      7   ; floating point preserved register
```

Data representation and argument passing

This standard does not describe the layout in store of records, arrays and so forth, used by languages (such as C) on the ARM. For this information, consult the appropriate documentation of each specific language. The procedure call standard is defined in terms of *n* word-sized arguments being passed from the caller to the callee, and a single word or floating point result that is passed back by the callee. For a detailed description of how these facilities are used to implement open array arguments, structure arguments, structure results, etc, also consult the appropriate documentation for each specific language.

Control arrival

We consider the passing of N (>= 0) actual argument words to a procedure which expects to receive either exactly N argument words or a variable number V (>= 1) of argument words (it is assumed that there is at least one argument word which indicates in a language-implementation-dependent manner how many actual argument words there are, eg by using a format string argument, a count argument, or an argument-list terminator).

At the instant when control arrives at the target procedure, the following statements should be true. (For any *m*, if a statement is made about *argm* and *n<m*, then the statement can be ignored.)

- *arg1* is in `a1`
- *arg2* is in `a2`
- *arg3* is in `a3`
- *arg4* is in `a4`
- *arg5* is at `[sp]`
- for all m > 5, *argm* should be at `[sp, #4*(m-5)]`
- `fp` contains 0 or points to a backtrace structure, as described in the next section
- the values in `sp` and `fp` are all multiples of four
- `lr` contains the `pc+psw` value that should be restored into `r15` on exit from the procedure. This is known as the *return link value* for this procedure call
- `pc` contains the entry address of the target procedure.

Now, let us call the lower limit to which sp may point *in this stack chunk* SP_LWM (Stack-Pointer Low Water Mark). Then, space between sp and SP_LWM shall be (or shall be on demand) readable, writable memory which can be used by the called procedure as temporary workspace and overwritten with any values before the procedure returns.

```
sp >= SP_LWM + 256
```

This condition guarantees that a stack extension procedure, if used, shall have a reasonable amount (256 bytes) of work space available to it, probably sufficient to call two or three procedure invocations further.

Return Control

At the instant when the return link value for a procedure call is placed in the `pc+psw`, the following statements should be true:

- `fp`, `sp`, `v1`, `v2`, `v3`, `v4`, `v5`, `v6`, `f4`, `f5`, `f6` and `f7` should contain the same values as they did at the instant of the call.
- If the procedure returns a word-sized result, R, which is not a single-precision floating point value, then `R` should be in `a1`.
- If the procedure returns a single or double precision floating point result, fpr, then fpr should be in `f0`.

Notes

- The requirements of C preclude the passing of floating point arguments in floating point registers.
- The requirement to pass a variable number of arguments to a procedure (as in old-style C) precludes the passing of floating point arguments in floating point registers (as the ARM's fixed point registers are disjoint from its floating point registers). However, if a callee is defined to accept a fixed number K of arguments and its interface description declares it to accept exactly K arguments of matching types, then it is permissible to pass the first four floating point arguments in floating point registers f0-f3.
- The values of `a2`, `a3`, `a4`, `ip` and `lr` are not defined at the instant of return. The maintenance of the registers `v1` to `v6`, however, suggests that this should be thought of as a 'callee-saving' standard.
- The values of the Z, N, C and V flags are loaded from the corresponding bits in the return link value on procedure return. This means, in the case where a procedure is called using a BL instruction, that these flag values will be preserved across the call. Note that the flag values in `lr` at the instant of entry must be instated, rather than preserving the flag values from the instant of entry. This is so that tail-folding procedure exit can work properly.
- This procedure call standard does not define the values of `fp` and `sp` at arbitrary execution moments during the evaluation of a procedure, only at the instants of call and return. It should be noted that further standards may be relevant when working with particular operating systems, in order to aid event handling and debugging. In general you are strongly advised to preserve fp and sp at all times.
- Read also the comments in the `aupcs(7)` manual page.

The stack backtrace structure

At the instant of an external procedure call, the value in `fp` is zero or it points to a data structure that gives information about the sequence of outstanding procedure calls. The Stack Backtrace Data Structure format is shown below.

```
fp points to here: |  save mask pointer |[fp]
          |  return link value |   [fp, #-4]
          |  return sp value   |   [fp, #-8]
          |  return fp value   |   [fp, #-12]
         [|  saved v6 value    |]
         [|  saved v5 value    |]
         [|  saved v4 value    |]
         [|  saved v3 value    |]
         [|  saved v2 value    |]
         [|  saved v1 value    |]
         [|  saved f7 value    |]  three words
         [|  saved f6 value    |]  three words
         [|  saved f5 value    |]  three words
         [|  saved f4 value    |]  three words
```

The format shows between four and 22 words of store, with those words higher on the page being at higher addresses in memory. The values shown in square brackets are entirely optional, and the presence of any does not imply the presence of any other. The floating point values are in extended format and occupy three words each. At the instant of the procedure call, all of the following statements about this structure must be true:

- The `return fp value` (shown above) is either 0 or contains a pointer to another stack backtrace data structure of the same form. Each of these corresponds to an active, outstanding procedure invocation. The same statements

listed here are just as true about this next stack backtrace data structure as they are for the current one. Thus, the statements hold true for each structure in the chain.

- The `save mask pointer` value, when bits 0, 1, 26, 27, 28, 29, 30, 31 have been cleared, points 12 bytes beyond a word known as the *return data save instruction*.
- The return data save instruction is a word that corresponds to an ARM instruction of the following form:

```
stmdb   sp!,    {
                [v1,]   [v2,] [v3,] [v4,] [v5,]         [v6,]
                fp, ip, lr, pc}
```

  Note the square brackets in the above form denote optional parts: thus, there are 64 possible allowable values for the return data save instruction, corresponding to the following bit patterns:

```
1110 1001 0010 1101 1101 10xx xxxx 0000
 E    9    2    D    D   (8)
```

  The highest six bits of the lowest ten bits represent the registers, ie if bit *n* is set, then register n will be transferred.

  The optional parts `[v1]`, `[v2]`, `[v3]`, `[v4]`, `[v5]` and `[v6]` in this instruction correspond to those optional parts of the stack backtrace data structure that are present such that: for all *m*, if `[vm]` is present then so is `[| saved vm value |]`, and if `[vm]` is absent then so is `[| saved vm value |]`. This is just as though the stack backtrace data structure was formed by the execution of this instruction, following the loading of `ip` from `sp` – as is very probably the case.

The sequence of up to four instructions following the return data save instruction decides if the saved floating point registers are present. The four instructions that are allowed in this sequence are:

```
STFE    f7, [sp, #-12]! ; [f7], 11101101 01101101 01110001 00000011
STFE    f6, [sp, #-12]! ; [f6], 11101101 01101101 01100001 00000011
STFE    f5, [sp, #-12]! ; [f5], 11101101 01101101 01010001 00000011
STFE    f4, [sp, #-12]! ; [f4], 11101101 01101101 01000001 00000011
```

Any or all of these instructions may be missing, and any deviation from this order or any other instruction terminates the sequence.

The optional instructions `[f4]`, `[f5]`, `[f6]` and `[f7]` in this sequence correspond to those optional parts of the stack backtrace data structure that are present such that: for all *m*, if `[fm]` is present then so is `[| saved fm value |]`, and if `[fm]` is absent then so is `[| saved fm value |]`. This is just as though the stack backtrace data structure was formed by the execution of this sequence, as is probably the case.

- At the instant when *procedure a* calls *procedure b*, the stack backtrace data structure pointed at by `fp` contains exactly those elements `[v1]`, `[v2]`, `[v3]`, `[v4]`, `[v5]`, `[v6]`, `[f4]`, `[f5]`, `[f6]`, `[f7]`, `fp`, `sp` and `pc` which must be restored into the corresponding ARM registers in order to cause a correct exit from *procedure a*, albeit with a junk result.

Notes

This example suggests what the entry and exit sequences for a procedure are likely to be. Though not defined in terms of the following sequences because that would be unnecessarily restrictive, the entry sequence to a typical procedure might be expected to look something like:

```
mov         ip, sp
stmdb       sp!, {argRegs, workRegs, fp, ip, lr, pc}
sub         fp, ip, #4
```

The corresponding exit sequence would be:

```
ldmdb       fp, {workRegs, fp, sp, pc}^
```

fp should be valid at all times. Signals are not blocked – therefore the

```
sub fp,ip,#4
```

must occur after the `stfe` instruction.

Many apparent idiosyncrasies in the standard may be explained by efforts to make the entry sequence work smoothly. This example above is neither complete nor mandatory (making arguments contiguous for C, for instance, requires a slightly different entry sequence).

The 'workRegs' registers mentioned above correspond to as many of `v1` to `v6` that this procedure needs in order to work smoothly. At the instant when *procedure a* calls any other procedure, those workspace registers not mentioned in *procedure a*'s return data save instruction will contain the values that they contained at the instant that *a* was entered. Furthermore, the registers `f4-f7` not mentioned in the floating point save sequence following the return data save instruction will also contain the values that they contained at the instant that *procedure a* was entered.

2.4 Defined bindings

These bindings of the ARM procedure-call standard are used by:

- RISC iX kernels running in ARM SVC mode.

  The call-frame register bindings are:

```
sl      RN      10  ; stack limit / stack chunk handle
                    ; unused by RISC iX applications
fp      RN      11  ; frame pointer
ip      RN      12  ; used as temporary workspace
sp      RN      13  ; lower end of current stack frame
```

  The invariants sp > ip > fp have been preserved allowing symbolic assembly code (and compiler code-generators) written in terms of register names to be ported between APCS-R, APCS-U and APCS-A merely by relabelling the call-frame registers provided:

- when call-frame registers appear in LDM, LDR, STM and STR instructions they are named symbolically, never by register numbers or register ranges
- no use is made of the ordering of the four call-frame registers (eg in order to load/save fp or sp from a full register save).

APCS-U constraints on sl

For RISC iX applications and RISC iX kernels. In this binding of the APCS the user-mode stack auto-extends on demand so sl is unused and there is no stack-limit checking.

In kernel mode, sl is reserved to Acorn.

Notes on APCS bindings

In all future supported bindings of APCS sp shall be bound to r13. In all supported bindings of APCS the invariant sp > ip > fp shall hold. This means that the only other possible binding of APCS is APCS-M:

```
sl      RN      12      ; stack limit / stack chunk handle
                        ; unused by RISC iX applications
fp      RN      10      ; frame pointer
ip      RN      11      ; used as temporary workspace
sp      RN      13      ; lower end of current stack frame
```

Further restrictions in SVC mode and IRQ mode

There are some consequences of the ARM's architecture which, while not formally acknowledged by the ARM Procedure Call Standard, need to be understood by implementors of code intended to run in the ARM's SVC and IRQ modes.

An IRQ corrupts r14_irq, so IRQ-mode code must run with IRQs off until r14_irq has been saved. Acorn's preferred solution to this problem is to enter and exit IRQ handlers written in high-level languages via hand-crafted 'wrappers' which on entry save r14_irq, change mode to SVC, and enable IRQs and on exit restore the saved r14_irq (which restores IRQ mode and the IRQ-enable state). Thus the handlers themselves run in SVC mode, avoiding this problem in compiled code.

Both SWIs and aborts corrupt r14_svc. This means that care has to be taken when calling SWIs or causing aborts in SVC mode.

In high-level languages, SWIs are usually called out of line so it suffices to save and restore r14 in the calling veneer around the SWI. If a compiler can generate in-line SWIs, then it should, of course, also save and restore r14 in-line, around the SWI, in case the code has to run in SVC mode.

An abort in SVC mode may be symptomatic of a fatal error or it may be caused by page faulting in SVC mode. (Acorn expects SVC-mode code to be 'correct', so these are the only options.). Page faulting can occur because an instruction needs to be fetched from a missing page (causing a prefetch abort) or because of an attempted data access to a missing page. The latter may occur even if the SVC-mode code is not itself paged (consider an unpaged kernel accessing a paged user-space).

A data abort is completely recoverable provided r14 contains nothing of value at the instant of the abort. This can be ensured by:

- saving R14 on entry to every procedure and restoring it on exit;
- not using R14 as a temporary register in any procedure;
- avoiding page faults (stack faults) in procedure entry sequences.

  A prefetch abort is harder to recover from and an aborting BL instruction cannot be recovered. So:

- special action has to be taken to protect page faulting procedure calls.

For Acorn C, r14 is saved in the second or third instruction of an entry sequence. Aligning all procedures at addresses which are 0 or 4 modulo 16 ensures that the critical part of the entry sequence cannot prefetch-abort. A compiler can do this by padding all code sections to a multiple of 16 bytes in length and being careful about the alignment of procedures within code sections.

Data-aborts early in procedure entry sequences can be avoided by using a software stack-limit check like that used in APCS-R.

Finally, the recommended way to protect BL instructions from prefetch-abort corruption is to precede each BL by a MOV ip, pc instruction. If the BL faults, the prefetch abort handler can safely overwrite r14 with ip before resuming execution at the target of the BL. If the prefetch abort is not caused by a BL then this action is harmless, as r14 has been corrupted anyway (and, by design, contained nothing of value at any instant a prefetch abort could occur).

Chapter 3 – C compiler (cc)

C compiler (cc)

3.1 Introduction

This chapter outlines the main features of the RISC iX C compiler, cc, for RISC iX systems using the Acorn RISC Machine (ARM) processor.

The RISC iX C compiler accepts the same dialect of the C programming language as the 4.3BSD UNIX Portable C Compiler (pcc). This dialect of C is essentially that described in Kernighan and Ritchie's original book, *The C Programming Language, 1st Edition*. The RISC iX C compiler can also accept ANSI draft standard C (our policy is one of gradual upgrade to the latest version of the draft standard, until the draft standard is finalised).

This chapter does not act as an introduction to C. It is intended to be an explanation of how the RISC iX C Compiler meets, and differs from, the 4.3BSD Berkeley UNIX C compiler standard and the Kernighan and Ritchie standard.

If you need more information of a more introductory nature, we recommend the following books:

- *The C programming language, 2nd edition* by BW Kernighan and DM Ritchie, published by Prentice-Hall, Englewood Cliffs, NJ, USA. This is the original C 'bible', useful for a description of pcc-style C.
- *A C Reference Manual, 2nd edition* by Samuel P Harbison and Guy L Steele Jr, published by Prentice-Hall, Englewood Cliffs, NJ, USA. This is a very thorough reference guide to C, including a useful amount of information on the proposed ANSI draft for C.
- *C Language* by Friedman Wagner-Dobler, published by Pitman, 128 Long Acre, London UK.
- *C Puzzle Book* by Alan R Seuer, published by Prentice-Hall, Englewood Cliffs, NJ, USA.
- *The X/OPEN Portability Guide*, published by Elsevier Science Publishers BV, 1000 BZ, Amsterdam, Netherlands.

3.2 Running the compiler

This section describes how to run the compiler and control various aspects of its operation.

Naming conventions

The RISC iX C compiler, in common with many other C compilers, uses naming conventions to identify the four classes of file involved in the compilation and linking process. For example, the suffix `.c` denotes C source files.

The following section summarises the conventions for referring to source, include, object and program files.

Source files

A file `foo.c` will be looked for in the current directory.

Include files

The way in which the compiler deals with included files depends on whether the filename in the `#include` directive is between angled brackets < > or double quotation marks " ". The former case implies that the file is a 'system one', the latter that it is a user file. Simply put, system include files are searched for in the system include path. User include files are sought in the current directory and then in the system include path.

In both cases, an intermediate set of directories to search may be specified using the `-I` command-line flag. Much more information about include file processing may be found in the description of the `-I` flag below.

Object files

The object files created by the compiler are stored with the suffix '`.o`' in the currently selected directory. For example `foo.o`.

Compilation list files

If the –list keyword is specified, then a file containing the compilation listing for each compiled source file is created with the suffix "`.lst`".

Library files

The default link step activated by the compiler searches for libraries in the system library path.

Program files

The result of linking the compiled versions of the source files given in the command line with the libraries is an executable program file. This is named a.out by default, but can be given any other name by using the –o flag (see below).

Compiling a simple program

The RISC iX C compiler is called `cc`. To run it, type:

`cc` *[options] ...filenames...*

The *options* allow you to control the compilation by, for example, over-riding default names. Often, just the list of one or more filenames is all that is required following the `cc` command. For example, suppose you have created the following program:

```
#include <stdio.h>
int main()
{
      printf("Hello world\n");
      return 0;
}
```

and store it as `hello.c`.

To compile it, you enter the command

```
cc hello.c
```

Assuming that everything is correct, this compilation produces the executable file `a.out`. When the executable file `a.out` is run, it produces the output:

```
Hello world
```

3.3 Compiler options

You can control many aspects of the compiler's operation by appending options to the command `cc`. All options are prefixed by the minus sign `-`.

Options come in two forms. The first are keywords. These are multiple-character options and control Acorn-specific aspects of the compiler. Keywords are recognised in upper or lower case. The second form of option is the flag. A flag is a single letter, upper or lower case. Flags are case dependent.

The keyword options are:

`-help` Gives a short description of the compiler's command syntax.

`-list` Create a listing file. This consists of lines of source interleaved with error and warning messages. Finer control over the contents of this file may be obtained using the `-f` flag (see below).

`-ansi` Compile ANSI standard C source to recent draft ANSI specifications.

The flag options are listed below. Some of these are followed by an argument. The descriptions are divided into several sections, so that flags controlling related aspects of the compiler's operation are grouped together.

Controlling the linker

`-c` Suppresses the loading phase of the compilation, and forces object file(s) to be produced with a '`.o`' suffix.

`-Ldir` This specifies an additional directory in which the `ld` command will search for library files.

`-l`*<library>* This causes `ld` to load the library whose name is `lib`*library*`.a` from the library load path.

Controlling the preprocessor

The `-I` and `-j` flags control the search paths used when the compiler is looking for included files. Before describing these flags, we explain the way in which the search paths are used.

The list of paths comprises three separate sections. In order, these are the directory containing the current source file, any directories given by `-I` flags (in the order they appear on the command line), and finally the system include path. In the absence of any `-j` flag on the command line, the system include path points to the default system library, `/usr/include`.

When a directive of the form:

`#include` *<filename>*

is encountered, the compiler performs the following actions. First, the list of directories described above is searched, omitting the first one, the directory in which the source file is located. So first, any directories given by `-I` flags are searched, then the system include path. The filename that is actually looked for is formed by the concatenation of the directory name and the include filename, suitably separated. Eg, if the name in angled brackets was `expr.h` and the option `-Isyshdrs` flag was given. The compiler would look for `syshdrs/expr.h`.

If none of the `-I` directories yields the file, then the system include path is searched.

`-I`*pathname* This adds the specified pathname to the list of directories which are searched for #include files. The directories are searched in the order in which they are given in `-I` options. The preprocessor first searches for #include files in the directory containing the *sourcefile*, then in the directories named with the `-I` option and finally in the system include path list of directories (`/usr/include`).

`-j` *dirs* This overrides the system include path with the list of directories which follows the flag. The directories are separated by commas.

`-E` If this flag is specified, only the macro preprocessor phase of the compiler is executed. The output from the preprocessor is sent to the standard output. It can be redirected to a file using the usual methods. By default, comments are stripped from the output, but see the next flag.

`-C` This is used in conjunction with `-E` above. It reinstates comments, so they appear in the output produced by the preprocessor. Note that it is different from the `-c` flag, which is used to suppress the link operation.

`-M` Generate a 'makefile' style list of dependencies on stdout.

`-R` Make string-literals read-only (and hence, sharable).

Controlling code generation

The options described in this section control in some way the code produced by the compiler.

`-g` This flag is used to specify that additional debugging symbol tables for use by `dbx(1)` should be generated. Some optimisations are turned off by this option. A debugging library (`-lc -g`) is used instead of the standard library.

`-o` *file* This flag specifies the name of a file in which the output of the linker should be stored. It overrides the default, which is `a.out`. The existing contents of `a.out` remain undisturbed.

`-p` This flag causes the compiler to generate code which allows profiling of the program to take place. The code counts the number of times each routine is called. If loading takes place, replace the standard start-up routine by one which automatically calls `monitor(3)` at the start and arranges to write out a `mon.out` object program. An execution profile can then be generated by use of `prof(1)`. A profiling library is used instead of the standard C library (`-lc -p`).

`-pg` Causes the compiler to produce counting code in the manner of `-p`, but invokes a run-time recording mechanism that keeps more extensive statistics and produces a `gmon.out` file at normal termination. Also a profiling library is searched, in lieu of the standard C library. An execution profile can then be generated by use of `gprof(1)`.

`-S` If this flag is specified, no object code is generated and no attempt is make to link it. Instead, an assembly listing of the compiled code produced is written to a file called `file.s`, where `file` is the name of the source file (stripped of any directories or suffixes).

`-O` Turns on any user selectable optimisations. Usually optimisations are enabled, so `-o` is ignored. But this varies from release to release.

`-D`*sym*
`-D`*sym*`=`*value* Define *sym* as a preprocessor macro, as if by a line `#define` *sym* at the head of the source file. If no definition is given, the name is defined as 1.

`-U`*sym* Undefine *sym*, as if by a line `#undef` *sym* at the head of the source file. This may be used to cancel the effect of otherwise predefined symbols, eg arm. (The macro arm is predefined, and has the value 1.)

Controlling warning messages

The `-w` option controls the suppression of warning messages. Usually the compiler is very free with its warnings, as this tends to indicate potential portability or other problems. However, too many such messages can be a nuisance in the early stages of developing a program, so they may be disabled.

`-w` All warnings are suppressed.

Differences between the RISC iX C compiler and the BSD C compiler.

`-go` This is not supported, (no support for `sdb`).

`-R` Makes string literals read-only and sharable, rather than making all initialised variables read-only and sharable (by putting them in the text segment).

`-f` Is not supported. Float expressions are always converted to double (but register float variables work anyway).

`-B` and `-t` Are not supported. There are no separate compiler passes and there are no substitutes.

3.4 Compiler compatibility

This section discusses the differences between the RISC iX C compiler and the language as defined by the 4.3BSD Berkeley UNIX C Compiler. The C compiler will accept (Berkeley) UNIX-compatible C, as defined by the implementation of the Portable C Compiler (pcc) subject to the restrictions noted below.

In essence, C is Kernighan and Ritchie C, as defined in the book *The C Programming Language*, by B. Kernighan and D. Ritchie (K&R), with a small number of extensions and clarifications of language features that the book leaves undefined.

Language and preprocessor differences

The RISC iX C Compiler accepts K&R C , but it does not accept many of the old-style compatibility features, the use of which has been warned against for many years. Differences are listed briefly below:

- The following, dangerous, non-portable code, is found in some UNIX tools pre-dating UNIX Version 7:

```
struct {int a, b;};
double d;
d.a = 0;
d.b = 0x....;
```

  This is accepted by UNIX compilers and may cause problems when porting old code.

- enums are less strongly typed than is usual under Portable C Compilers. enum is a non-K&R extension to C which has been standardised by ANSI somewhat differently from the usual pcc implementation.
- The compiler permits the use of the ANSI `...` notation which signifies that a variable number of formal arguments follow.
- With the exception of enums, the compiler's type checking is generally stricter than pcc's (much more akin to lint's). In writing the RISC iX C compiler, we have attempted to strike a balance between generating too many warnings when compiling known, working code, and warning of poor or non-portable programming practices. Many compilers silently compile code which has no chance of executing in just a slightly different environment. We have tried to be helpful to those who must port C among machines in which the following vary:
  - the order of bytes within a word (eg little-endian ARM, VAX, Intel versus big-endian Motorola, IBM370),
  - the default size of `int` (four bytes versus two bytes in many PC implementations),
  - the default size of pointers (not always the same as `int`),
  - whether values of type `char` default to `signed` or `unsigned char`,
  - the default handling of undefined and implementation-defined aspects of the C language.

If the verbosity of `cc` is found undesirable then all warnings can be turned off by using the `-w` command-line flag.

- The compiler's preprocessor is believed to be equivalent to UNIX's cpp, except for the points listed below. cpp is defined only by its implementation, and although equivalence has been tested over a large body of UNIX source code, completely identical behaviour cannot be guaranteed. Some of the points listed below only apply when the `-E` option is used with the `cc` command.
  - There is a different treatment of whitespace sequences (benign).

- \<nl> is processed by `cc -E`, but passed by `cpp` (making lines longer than expected; `cc -E` only).
- cpp breaks long lines at a token boundary; `cc -E` doesn't, this may break line-size constraints when the source is later consumed by another program (`cc -E` only).
- The handling of unrecognised # directives is different (this is mostly benign).
- Literal control characters cause `cc` to issue a warning.
- The ANSI `#elif` construct is permitted.
- The ANSI `#error` construct is permitted.
- The ANSI `#pragma` construct is permitted.
- The ANSI `#if defined ...` construct is permitted.
- The following ANSI pre-defined macros can be used (and can be redefined):

  `__DATE__, __FILE__, __LINE__, __TIME__.`

Lexical and syntactic differences

- The keywords `asm`, `entry` and `fortran` are ignored.
- The ANSI modifiers `f` or `F` and `u` or `U` can be used in conjunction with `l` or `L` to denote floating, unsigned, and long values, respectively. For example: `2lf` (double 2.0); `3u` (unsigned 3); `4l` (long 4).
- The ANSI implicit concatenation of adjacent strings is permitted. For example `'Hello ' 'World'` is treated as `'Hello World'`.
- The ANSI escape sequences `\a` and `\xnn` can be used within strings and character constants to denote the ASCII BEL character and a hexadecimal character code respectively. As `\a` and `\x` are meaningless in pcc-style C, existing usage can only be accidental.
- ANSI function prototypes may be used. These give precise information about a function's arguments and result type. Such prototypes may include use of the ANSI `...` to denote a variable number of arguments following. For example:

  ```
  extern int funct(int a, float b, ...);.
  ```

- Use of the ANSI generic pointer type `void*` is permitted.
- Compound assignment operators where the = sign comes first are accepted (with a warning) by some compilers. An example is =+ instead of +=. Acorn C does not allow this ordering of the characters in the token. The symbols may not contain embedded whitespace. For example: + = is faulted.
- The = sign before a `static` initialiser was not required in some very old (pre-K&R C) compilers. Acorn C does not support this syntax.

Semantic differences

- An identifier declared with storage class `extern` has normal scope. If such an identifier is not declared at the outermost level then it does not have file scope.
- A function's formal arguments are considered to be defined at the head of the outermost block of the function. Thus they cannot be redefined within this block.
- A `struct`, `union` or `enum` tag defined within a function's formal argument list is considered to be defined in the block containing the function. Such tags may not be redefined later in following formal argument lists.

- A struct or union field name may only be used to select a component from an expression of the appropriate type. The archaic and abominable:
  `struct {int a, b;}; double d; d.b = 0;...` is faulted.
- New name types defined using `typedef` may not have modifiers applied to them. Hence, code such as: `typedef int tl;unsigned tl a; ...` is faulted.
- The type of a variable may not be multiply, defined or redefined. Thus, code such as:

  `long x[3]; int x[3];...` is faulted.
- The braces around a struct of an array may not be partially omitted. For example, consider:

  ```
  typedef struct{ struct{ int a,b; } c[3]; } T;
  ```

  The following initialisers are accepted;

  ```
  static T v1 = { 1,2,3,4,5,6 };
  ```

  and

  ```
  static T v2 = {{{1,2},{3,4},{5,6}}};
  ```

  however the initialiser

  `static T v3 = {{1,2},{3,4},{5,6}}` is faulted.
- As in ANSI C, union types may be initialised. The initialisation applies to the first arm of the union.
- As in ANSI C, the extended precision floating point type `long double` is available.
- Bitfields are packed differently. The packing of fields described as `char`, `short` or `long` is unaffected by those qualifiers (which is different to the behaviour of 4.3BSD UNIX's `cc`). Acorn's packing rule is to pack all the bitfields (regardless of type) into a structure as tightly as possible.

General warnings

The following general warnings outline areas where subtle differences between the Acorn C compiler and the 4.3BSD C compiler may cause problems.

- Large uninitialised `static` data structures are written to the local object file. They are not defined as local bss. Thus, the top level definition; `static int v[1000];` causes an object file to be 4000 bytes longer (ie 1000 * 4) then the definition `int v[1000]`.
- When compiling source code the Acorn compiler (like the 4.3BSD C compiler) makes the types of `char`, `short`, `int`, `long` and bitfields signed by default. However, when compiling in ANSI-C mode the types of `char` and bitfields become unsigned by default. In addition to this the result of the built-in function `sizeof` is of type `int` in pcc mode and unsigned in ANSI C mode. These differences may give rise to problems with existing source code.

3.5 Compiling and linking

As already shown, to compile and link the simple program shown above you would type:

```
cc hello.c
```

This produces the executable program `a.out`. To produce a program with a different name, you would use the `-o` option, eg:

```
cc -o greeting hello.c
```

This time the linker would produce a program that you could run using the command `greeting`.

When writing programs that use several source files, you may want to compile them selectively and perform the link as a separate step. For example, if a program consists of the files `e1.c`, `e2.c` and `e3.c`, and you have just edited `e2.c`, you may want to compile this, then link the object file with the two other files:

```
cc -c expr2.c

ld -o expr expr1.0 expr2.0 expr3.0
```

Alternatively,

```
cc -o expr expr1.o expr2.c expr3.o
```

does the trick!

See the linker documentation for more details on linking. Consider using the 'make' utility to maintain even the simplest multi-file programs.

To compile several different source programs and link them all together into one executable file, list all the filenames separated by spaces. The name of the executable program is taken from the first filename given unless `-o` *progname* is used.

For example, the command:

```
cc mainprog.c util.c extra.c
```

compiles the sources `mainprog.c`, `util.c` and `extra.c` into the object files `mainprog.o`, `util.o` and `extra.o`, and then links all three object files together with the standard library to produce the executable program `a.out`.

3.6 Implementation details

This section gives details of those aspects of the compiler which Kernighan and Ritchie and/or ANSI identify as implementation-defined, and details of some other points of interest to programmers. They are grouped by subject.

Identifiers

Identifiers can be of any length. They are truncated by the compiler to 256 characters, all of which are significant.

The source character set is determined by the host operating system. Upper and lower case characters are distinct in all identifiers, both internal and external.

Data elements

The sizes of data elements are as follows:

Type	Size in bits	
`char`	8	
`short`	16	
`int`	32	
`long`	32	
`float`	32	
`double`	64	
`long double`	64	(subject to future change)
`all pointers`	32	

Integers are represented in two's complement form.

Data items of type `char` are signed by default, though they may be explicitly declared as `signed char` or `unsigned char`.

Floating point quantities are stored in the IEEE format. In double and long double quantities, the word containing the sign, the exponent and the most significant part of the mantissa is stored at the lower machine address.

Structured data types

The details of layout of the components of structured data types is implementation dependent. The following points apply to the RISC iX C Compiler:

- Structures are aligned on word boundaries.
- Structures are arranged with the first-named component at the lowest address.
- `Char` components are placed in adjacent bytes.
- `Short` components are aligned at even-addressed bytes.
- All other arithmetic type components are word-aligned, as are pointers and the first bitfield of each contiguous sequence of bitfields.
- The only valid types for bitfields are `int`, `long`, short, `char` (either `signed` or `unsigned`).
- A bitfield of type `int` is treated as signed by default.
- A bitfield must be contained within the 32 bits of an `int`.

Pointers

The following points apply to pointer types:

- Adjacent bytes have addresses which differ by one.
- The macro `NULL` expands to the value 0, with a pointer type.
- Casting between integers and pointers results in no change of representation.
- The compiler warns of casts between pointers to functions and pointers to data.

Pointer subtraction

When two pointers are subtracted, the difference is obtained as if by the expression:

`((int)a - (int)b) / (int)sizeof(`*type pointed to*`)`

If the pointers point to objects whose size is no greater than four bytes, word alignment of data ensures that the division will be exact in all cases. For longer types, such as doubles and structures, the division may not be exact unless both pointers are to elements of the same array. Moreover the quotient may be rounded up or down at different times, leading to potential inconsistencies.

Arithmetic operations

The compiler performs all of the usual arithmetic conversions.

The following points apply to operations on the integral types:

- All signed integer arithmetic uses a two's complement representation.
- Bitwise operations on signed integral types follow the rules which arise naturally from two's complement representation.
- Right shifts on signed quantities are arithmetic.
- Any quantity which specifies the amount of a shift is treated as an unsigned 8-bit value.
- Any value to be shifted is treated as a 32-bit value.
- Left shifts of more than 31 give a result of zero.
- Right shifts of more than 31 give a result of zero from an unsigned or positive signed value, –1 from a negative signed value.
- The remainder on integer division has the same sign as the divisor.
- If a value of integral type is truncated to a shorter signed integral type, the result is obtained by masking the original value to the length of the destination and then sign extending.
- Conversions between integral types never cause exceptions to be raised.
- Integer overflow does not cause an exception to be raised.
- Integer division by zero causes an exception to be raised.

The following points apply to operations on floating types:

- The ARM's floating point registers are wider than stored floating point numbers, so that some values may be computed to a slightly higher precision than the stated limits imply.
- When a `double` or `long double` is converted to a float, rounding is to the nearest representable value.
- Conversions from floating to integral types cause exceptions to be raised only if the value cannot be represented in a `long int` (or `unsigned long int` in the case of conversion to an `unsigned int` type).
- Floating point underflow is not detected; any operation which underflows returns zero.
- Floating point overflow causes an exception to be raised.
- Floating point divide by zero causes an exception to be raised.

Expression evaluation

The compiler performs the usual arithmetic conversions (promotions) set out in the ANSI draft standard reference before evaluating any expression.

- The compiler may re-order expressions involving only associative and commutative operators, even in the presence of parentheses.

- Between sequence points, the compiler may evaluate expressions in any order. Thus the side effects of expressions between sequence points may occur in any order.
- Similarly, the compiler may evaluate function arguments in any order; moreover, this order may change from release to release.

Implementation limits

There are certain minimum translation limits which a compiler must cope with; you should be aware of these if you are porting applications to other compilers. A summary is given here. The mem limit indicates that no limit is imposed other than that of available memory.

Description	ANSI Requirement	RISC iX C
Nesting levels of compound statements and Iteration/selection control structures	15	mem
Nesting levels of conditional compilation	6	mem
Declarators modifying a basic type	12	mem
Expressions nested by parentheses	127	mem
Significant characters		
– in internal identifiers and macro names	31	256
– in external identifiers	6	256
External identifiers in one source file	511	mem
Identifiers with block scope in one block	127	mem
Macro identifiers in one source file	1024	mem
Parameters in one function definition/call	31	50
Parameters in one macro definition/invocation	31	mem
Characters in one logical source line	509	no limit
Characters in a string literal	509	mem
Bytes in a single object	32767	mem
Nesting levels for `#included` files	8	mem
Case labels in a `switch` statement	255	mem
`atexit`-registered functions	32	33

3.7 Standard implementation definition

This section discusses aspects of the compiler which are implementation-defined and must be documented.

Identifiers

- 256 characters are significant in identifiers without external linkage. (Characters are letters, digits, and underscores.)
- 256 characters are significant in identifiers with external linkage. (Allowed characters are letters, digits, underscores but not extensions.)
- Case distinctions are significant in identifiers with external linkage.

Characters

The characters in the source character set are to ISO 8859-1 (Latin Alphabet), a superset of the standard ASCII character set. The printable characters are those in the range 32 to 126 and 160 to 255. All printable characters may appear in string or character constants, and in comments.

- The execution character set is identical to the source character set.
- There are four `chars` in an `int`. The bytes are ordered from least significant at the lowest address to most significant at the highest address.
- There are eight bits in a character in the execution character set. The bits are ordered such that the leftmost bit is most significant.
- Characters of the source character set in string literals and character constants map identically into characters in the execution character set.

- A character constant containing more than one character has the type `int`. Up to four characters of the constant are represented in the integer value. The first character contained in the constant occupies the lowest-addressed byte of the integer value; up to three following characters are placed at ascending addresses. Unused bytes are filled with the NUL character. This is not portable.
- A plain `char` is treated as signed.
- Escape codes are:

Escape sequence	Char value	Description
`\a`	7	Attention (bell)
`\b`	8	Backspace
`\f`	12	Form feed
`\n`	10	Newline
`\r`	13	Carriage return
`\t`	9	Tab
`\v`	11	Vertical tab
`\xnn`	nn	ASCII code in hexadecimal
`\nnn`	nnn	ASCII code in octal

Integers

The representations and sets of values of the integral types have been set out above in the section *Implementation details, Data elements*. Note also that:

- The result of converting an integer to a shorter signed integer, if the value cannot be represented, is as if the bits in the original value which cannot be represented in the final value were masked out, and the resulting integer sign-extended. The same applies when you convert an unsigned integer to a signed integer of equal length.
- Bitwise operations on signed integers yield the expected result given two's complement representation. No sign extension takes place.
- The sign of the remainder on integer division is the same as defined for the function `div()`.
- Right shift operations on signed integral types are arithmetic.

Floating point

The representations and ranges of values of the floating point types have been given above in *Implementation details, Data elements*. Note also that:

- When a floating point number is converted to a shorter floating point one, it is rounded to the nearest representable number.
- The properties of floating point arithmetic accord with IEEE 754.

Arrays and pointers

The ANSI draft standard specifies three areas in which the behaviour of arrays and pointers must be documented. The points to note are:

- The type `size_t` is defined as `unsigned int`.
- Casting pointers to integers and vice-versa involves no change of representation. Thus any integer obtained by casting from a pointer will be positive.
- The type `ptrdiff_t` is defined as `(signed) int`.

Registers

In the RISC iX C Compiler, you can declare up to six objects as having the storage class `register`. The valid types are:

- any integer type
- any pointer type
- any structure type which contains only bitfields and which is no more than one word long.

Note that other variables, not declared as `register`, may be held in registers for extended periods, and that `register` variables may be held in memory for some periods.

Structures, unions and bit-fields

The RISC iX C compiler handles structures in the following way:

- When a member of a union is accessed using a member of a different type, the resulting value can be predicted from the representation of the original type. No error is given.
- Structures are aligned on word boundaries. Characters are aligned in bytes, shorts on even numbered byte boundaries and all other types, except bit-fields, are aligned on word boundaries. Bit-fields are parts of `int`s, themselves aligned on word boundaries. However the bit-fields themselves are not aligned.
- A plain bit-field (declared as `int`) is treated as `signed int`.
- A bit-field which does not fit into the space remaining in an `int` is placed in the next `int`.
- The order of allocation of bit-fields within `int`s is such that the first field specified occupies the least significant bits of the word.
- Bit-fields do not straddle `int` boundaries.

Declarators

The number of declarators that may modify a basic type is limited only by available memory.

Statements

The number of `case` values in a `switch` statement is limited only by memory.

Preprocessing directives

- A single-character constant in a preprocessor directive cannot have a negative value.
- Quoted names for includable source files are supported.

Library functions

- The mathematical functions conform to the IEEE FP standard (see the `math(3)` manual page). Errors result in signals.
- The mathematical functions do not cause a signal on underflow range errors.
- The `setjmp()` function may be called in any expression context.
- The last line of a text stream does not require a terminating new line character.
- Text lines consisting solely of space characters followed by a newline character are read back exactly as written.
- No NUL characters are appended to a binary output stream.
- A zero-length file *does* exist.
- The validity of file names is defined by UNIX's filing system.
- The same file can be open many times for reading, writing or updating.

- Note also the following points about library functions:

`fprintf()` Prints `%p` arguments in hexadecimal format as if a precision of eight had been specified. If the variant form is selected, the number is preceded by the character `@`.

`fscanf()` Treats `%p` arguments identically to `%x` arguments.

`fscanf()` Always treats the character `-` in a `%[` argument as a literal character.

`ftell()` Never reports failure.

`strcmp()` etc. The value returned by the memory and string-comparison functions is in the range –255 to +255. The value is negative if and only if the first string is lexically less than the second

For example, suppose s1 and s2 differ first in the third character. Then `strcmp (s1,s2)` differ first in the third character. Then:

```
strcmp (s1,s2) < 0 if s1[2] < s2[2].
```

3.8 Assembly language interface

Object code modules from the RISC iX C Compiler can be linked with those produced by `as`, provided that they observe the conventions of the *ARM UNIX Procedure Call Standard*. The following names are used in referring to ARM registers:

Register names

`a1`	R0	Argument 1, also integer result
`a2`	R1	Argument 2
`a3`	R2	Argument 3
`a4`	R3	Argument 4
`v1`	R4	Register variable
`v2`	R5	Register variable
`v3`	R6	Register variable
`v4`	R7	Register variable
`v5`	R8	Register variable
`v6`	R9	Register variable
`sb`	R10	Reserved to Acorn
`fp`	R11	Frame Pointer
`ip`	R12	Used as temporary workspace
`sp`	R13	Lower end of current stack frame
`lr`	R14	Link address on calls, or workspace
`pc`	R15	Program counter and processor status
`f0`	F0	Floating-point result
`f1`	F1	Floating-point work register
`f2`	F2	Floating-point work register
`f3`	F3	Floating-point work register
`f4`	F4	Floating-point register variable (must be preserved)
`f5`	F5	Floating-point register variable (must be preserved)
`f6`	F6	Floating-point register variable (must be preserved)
`f7`	F7	Floating-point register variable (must be preserved)

In this section, 'at `[r]`' means at the location pointed to by the value in register `r`; at `[r,#n]` refers to the location pointed to by `r+n`. This accords with `as` syntax.

Register usage

The following points should be noted about the contents of registers across function calls.

- Calling a function (potentially) corrupts the argument registers `a1` to `a4`, `ip`, `lr`, and `f0-f3`. The calling function should save the contents of any of these registers it may need.
- Register `lr` is used at the time of a function call to pass the return link to the called function; it is not necessarily preserved during or by the function call.
- The stack pointer `sp` is not altered across the function call itself, though it may be adjusted in the course of pushing arguments inside a function.
- Registers `v1` to `v6`, and the frame pointer `fp`, are expected to be preserved across function calls. The called procedure is responsible for saving and restoring the contents of any of these registers which it may need to use.
- `sp` should never point (numerically) above any valid data (because execution of a signal handler may use the stack at any time).
- `sb` is reserved to Acorn and should be preserved everywhere.

Control arrival

At a procedure call, the convention is that the registers are used as follows:

- `a1` to `a4` contain the first four arguments.
- `sp` points to the fifth argument; any further arguments will be located in succeeding words above `[sp]`.
- `fp` points to a backtrace structure.
- `lr` contains the value which should be restored into `pc` on exit from the called procedure.
- `pc` contains the entry address of the called procedure.

Passing arguments

All integral and pointer arguments are passed as 32-bit words. Floating point arguments are always passed as doubles (64 bits). These follow the memory representation of the IEEE single and double precision formats.

Arguments are passed as if by the following sequence of operations:

- Push each argument onto the stack, last argument first.
- Pop the first four words (or as many as were pushed, if fewer) of the arguments into registers `a1` to `a4`.
- Call the function, for example by the branch with link instruction: `BL` *functionname*.

In many cases it is possible to use a simplified sequence with the same effect (eg load three argument words into `a1-a3`).

If more than four words of arguments were passed, the calling procedure should adjust the stack pointer after the call, incrementing it by four for each argument word which was pushed and not popped.

Return link

On return from a procedure, the registers are set up as follows:

- `fp`, `sp`, `v1` to `v6` and `f4` to `f7` have the same values that they contained at the procedure call.

- Any result other than a floating point or a multi-word structure value is placed in register `a1`.
- A floating point result is placed in register `f0`.

Structure values returned as function results are discussed below.

Structure results

A C function which returns a multi-word structure result is treated in a slightly different manner from other functions by the compiler. A pointer to the location which should receive the result is added to the argument list as the first argument, so that a declaration such as the following:

```
s_type afunction(int a, int b, int c)
{
                s_type d;
                /* ... */
                return d;
}
```

is in effect converted to this form:

```
void afunction(s_type *p, int a, int b, int c)

{
                s_type d;
                /* ... */
                *p = d;
                return;
}
```

Any assembler-coded functions returning structure results, or calling such functions, must conform to this model in order to interface successfully with object code from the C compiler.

Storage of variables

The code produced by the C compiler uses procedure argument values from registers where possible; otherwise they are addressed relative to `fp`, as illustrated in the section *Examples* below.

Local variables, by contrast, are always addressed with positive offsets relative to `sp`. In code which alters `sp`, this means that the offset for the same variable will differ from place to place.

Examples

The following fragments of assembler code illustrate the main points to consider in interfacing with the C compiler. If you want to examine the code produced by the compiler in more detail for particular cases, you can request an assembler listing with the compiler option -S.

This is a function `gggg` which expects two integer arguments and uses only one register variable, `v1`. It calls another function `ffff`.

Example overleaf:–

```
@ Set up conventional names for some registers

a1              .req        r0
a2              .req        r1
a3              .req        r2
a4              .req        r3
v1              .req        r4
fp              .req        r11
ip              .req        r12
sp              .req        r13
lr              .req        r14
pc              .req        r15

                .text

                .ascii  "gggg"              @ Name of function,
                                            @ zero-terminated

                .ascii  "\0\0\0\0"          @ and padded to word
                                            @ boundary.

                .word   0xff000008          @ Name marker:
                                            @ length of name in low
                                            @ byte, 0xff in top byte.

@ Function Entry: save regs on stack as necessary.
                .global _gggg
_gggg:

                mov     ip, sp
                stmfd   sp!, {v1,fp,ip,lr,pc}
                sub     fp, ip, #4          @ Points to saved pc

                                            @ Main activity of function
                                            @.......
                add     v1, v1, #1          @ Use a register variable
                bl      _ffff               @ Call another function
                cmp         v1, #99         @ Rely on reg.var.aftercall

                                            @.......

@ Return: place result in a1 and restore registers
                mov     a1, #123
                ldmea   fp, {v1,fp,sp,pc}^

                .data                       @ Static data needed by
                                            @ this module

L__dataseg:
```

If required, the first four argument values can be stacked to be contiguous with argument five upwards as follows:

```
mov     ip, sp
stmfd   sp!, {a1,a2,a3,a4}   @ Save arg registers
stmfd   sp!, {v1,fp,ip,lr,pc}
sub     fp, ip, #20                     @ Point at saved pc
......
```

Chapter 4 – FORTRAN compiler (F77)

FORTRAN compiler (F77)

4.1 Introduction

This chapter describes the RISC iX Fortran Compiler (F77).

RISC iX Release 1.1 contains a beta-version of this compiler. The compiler is currently undergoing slight modifications and a full version, including full documentation will be released with the next release of RISC iX.

For information about the limitations of this beta-version of F77, read the release notice.

For a general paper about F77 read the 4.3BSD paper: *PS1:2 A portable Fortran 77 Compiler*. This is contained in the 4.3 Berkeley Software Distribution, UNIX Programmer's Supplementary Documents, Volume 1. Available from the European UNIX Users Group.

A beta-version of the F77 manual page follows. Consult the on-line manual page for late changes.

Name

F77 – Fortran 77 compiler

Synopsis

`f77 [ option ] ...` *file* ...

Description

F77 is the UNIX Fortran 77 compiler. It accepts several types of arguments:

Arguments whose names end with '`.f`' are taken to be Fortran 77 source programs; they are compiled, and each object program is left on the file in the current directory whose name is that of the source with '`.o`' substituted for '`.f`'.

Arguments whose names end with '`.F`' are also taken to be Fortran 77 source programs; these are first processed by the C preprocessor before being compiled by f77.

Arguments whose names end with '`.r`' or '`.e`' are taken to be Ratfor or EFL source programs respectively; these are first transformed by the appropriate preprocessor, then compiled by f77.

Arguments whose names end with '`.c`' or '`.s`' are taken to be C or assembly source programs and are compiled or assembled, producing a '`.o`' file.

The following options have the same meaning as in `cc(1)`. See `ld(1)` for load-time options.

`-c` Suppress loading and produce '`.o`' files for each source file.

`-g` Produce additional symbol table information for `dbx(1)` and pass the `-lg` flag to `ld(1)` so that on abnormal terminations, the memory image is written to file core. Incompatible with `-O`.

`-help` Output a summary manual entry for f77.

`-list` Generate a line-numbered compiler listing in a file with extension `.lst`.

`-o` *output* Name the final output file output instead of '`a.out`'.

`-p` Prepare object files for profiling, see `prof(1)`.

`-pg` Causes the compiler to produce counting code in the manner of `-p`, but invokes a run-time recording mechanism that keeps more extensive statistics and produces a `gmon.out` file at normal termination. An execution profile can then be generated by use of `gprof(1)`.

`-w` Suppress all warning messages. If the option is '`-w66`', only Fortran 66 compatibility warnings are suppressed.

`-D`*name=def*

`-D`*name* Define the name to the C preprocessor, as if by '`#define`'. If no definition is given, the name is defined as '`1`'. ('`.F`' suffix files only).

`-I`*dir* '`#include`' files whose names do not begin with '/' are always sought first in the directory of the file argument, then in directories named in `-I` options, then in directories on a standard list. ('`.F`' suffix files only).

`-O` Invoke an object-code optimiser. Incompatible with `-g`.

`-S` Compile the named programs, and leave the assembler-language output on corresponding files suffixed '`.s`'. (No '`.o`' is created.)

The following options are peculiar to f77:

`-i2` On machines which support short integers, make the default integer constants and variables short. (`-i4` is the standard value of this option). All logical quantities will be short.

`-m` Apply the M4 preprocessor to each '`.r`' file before transforming it with the Ratfor or EFL preprocessor.

`-onetrip`
`-1` Compile DO loops that are performed at least once if reached. (Fortran 77 DO loops are not performed at all if the upper limit is smaller than the lower limit.)

`-r8` Treat all floating point variables, constants, functions and intrinsics as double precision and all complex quantities as double complex.

`-strict` Compile only the core f77 language as specified in the ANSI standard X3.9-1978.

`-u` Make the default type of a variable 'undefined' rather than using the default Fortran rules.

`-v` Print the version number of the compiler as it executes.

`-C` Compile code to check that subscripts are within declared array bounds. For multi-dimensional arrays, only the equivalent linear subscript is checked.

`-F` Apply the C preprocessor to '`.F`' files, and the EFL, or Ratfor preprocessors to '`.e`' and '`.r`' files, put the result in the file with the suffix changed to '`.f`', but do not compile.

`-Ex` Use the string *x* as an EFL option in processing '`.e`' files.

`-Rx` Use the string *x* as a Ratfor option in processing '`.r`' files.

`-U` Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case except within character string constants.

Other arguments are taken to be either loader option arguments, or F77-compatible object programs, typically produced by an earlier run, or perhaps libraries of F77-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name '`a.out`'.

Programs compiled with f77 produce memory dumps in file core upon abnormal termination if the `-g` flag was specified during loading. If the environment variable *f77_dump_flag* is set to a value beginning with *y* or *n*, dumps for abnormal terminations are respectively forced or suppressed.

Files

`file.[fFresc]`	input file
`file.o`	object file
`a.out`	loaded output
`/usr/lib/f77comp`	compiler
`/lib/cpp`	C preprocessor
`/usr/lib/libF77.a`	intrinsic function library
`/usr/lib/libI77.a`	Fortran I/O library
`/usr/lib/libU77.a`	UNIX interface library
`/usr/lib/libm.a`	math library
`/lib/libc.a`	C library, see Section 3
`/usr/lib/libF77_p.a`	profiling intrinsic function library
`/usr/lib/libI77_p.a`	profiling Fortran I/O library
`/usr/lib/libU77_p.a`	profiling UNIX interface library
`/usr/lib/libm_p.a`	profiling math library
`/usr/lib/libc_p.a`	profiling C library, see Section 3
`mon.out`	file produced for analysis by `prof(1)`.
`gmon.out`	file produced for analysis by `gprof(1)`.

See also

`ld(1), ar(1), ranlib(1), dbx(1), intro(3f) efl(1), ratfor(1), prof(1), gprof(1), cc(1)`

Diagnostics

The diagnostics produced by f77 itself are intended to be self-explanatory. Occasional messages may be produced by the loader.

Bugs

This compiler is still somewhat experimental. It is intended to be compatible both with code conforming to the ANSI f77 standard X3.9-1978 and with code taking advantage of the extensions to this language provided by the BSD Unix F77 compiler. The standard part of the language is better specified and more easily tested than the extensions, so in general, bugs are more likely in these extensions (disabled using the `-strict` option). For details of current known bugs, see the release notes.

Chapter 5 – Writing device drivers

Writing device drivers

5.1 Introduction

This chapter outlines how a device driver fits into the kernel and provides sufficient information to permit a programmer with previous experience of UNIX device drivers to write and install a driver for an arbitrary device. It is expected that the majority of devices for which drivers will be written will be connected to the system hardware using the standard expansion card bus, and this chapter therefore concentrates on drivers of this type. Note that the term XCB is an acronym for eXpansion Card Bus and is often used within this chapter and the system source files.

5.2 Summary of device driver concepts

Device classification

Devices in UNIX are divided into two basic categories, conventionally referred to as *block* and *character* orientation. The distinction is not especially clear-cut or helpful in all cases, but the general idea is that a block device is one which stores significant volumes (from a few hundred kilobytes up to gigabytes) of data in a block-addressed, random access manner, and which may be used to contain a UNIX filing system structure; any other type of device is classed as a character device. All devices which are to be used as filing-system storage devices must have a block device interface; however it is common for such devices also to have a character interface. In this context the term *raw* is often used as a synonym for *character*.

Kernel interface

A device driver is a section of code which has a number of specific entry points which the kernel may call to implement particular operations. Tables of these entry points are maintained in a standard module, such that the kernel proper does not have to have the names of specific device driver functions built in, but rather calls them indirectly, via the tables. The term entry point here refers to external function names and potentially also external data items (eg, tty structures).

In RISC iX, beside the normal *bdevsw* and *cdevsw* entry point tables (for block and character devices respectively), there exists a table of interface information for devices (of any class) controlled via the expansion card bus. This table resides in a separate module, and is described more fully in the section below on the expansion card bus device interface.

It is conventional to name device driver entry points in a standard way, with a prefix related to the name of the driver, and the remainder of the identifier being fixed for each type of entry point. For example, a simple block device driver might use the prefix `df40_`; in this case the kernel interface routines in the driver might be called df40_open, df40_close, df40_strategy etc. Historically, such prefixes were limited to two characters (because of limitations of compiler/linker technology); however it is unnecessary to do this in the context of RISC iX, and also potentially awkward if names are not to clash. A prefix of four characters followed by an underscore character (_), is recommended (longer ones are possible but make source file layout messier, where several entry point names are used in initialising a devsw entry). If you intend to produce and distribute a device driver for RISC iX, please contact Acorn Customer Service to ensure that your chosen name is unique, since your driver may be linked into a kernel along with drivers from other independent sources.

Device numbers

Every device in the system is identified by a unique device number. This is defined in C terms by the typedef `dev_t`, and comprises a major number part and a minor number part. Standard macros `major(dev)` and `minor(dev)` are provided in the system include file `types.h`, to extract these components, each of which occupies eight bits.

The major number is used as an index into the *cdevsw* or *bdevsw* tables, according to whether the device is of character or block class respectively. Thus the major number selects the device driver out of the set of distinct device types in its class.

The kernel does not generally interpret the minor number itself: instead the driver is responsible for interpreting it, in any appropriate fashion. This varies from device to device, but there are general patterns. For block devices, it is common to partition a given physical storage device into sub-devices each containing some fraction of the total storage space. This partitioning may be either fixed by the driver code itself, or flexible, being defined for example by structures on the storage medium itself; the latter mechanism is preferred where possible. In this case, the least significant few bits of the minor number would be used to index a table of partitions on a given storage unit; the upper bits might be used to select a particular device controller and/or drive unit. See the manual page for the R140's internal ST506 disc, `st(4)`, as an illustration. For character devices the meaning of the minor number varies more substantially; however in the case of terminal (tty) drivers, it is normal (and parts of the kernel assume this as a default) to use the minor number as an index into an array of tty-control structures (eg, struct tty my_tty[NMY_TTY]), one for each distinct terminal supported by the driver. Another example of minor number usage is the standard RISC iX parallel printer driver – here bits in the minor number are used to control default settings of particular interface features: automatic form feed generation, tab-expansion, etc (see `lp(4)`).

Whatever scheme is implemented to interpret the minor number, it is important that it be used consistently throughout the driver. A common method is to define a set of macros or simple functions which given a minor number return particular sub-parts of it, typically by means of C's, & and >> operators, and use these macros or functions when determining partition number, tty sub-unit or whatever.

5.3 Summary of block device driver structure

A block device driver interfaces to the kernel through the bdevsw table (array). Each element of this array contains three main entry point fields and two subsidiary entry point fields; there are also three other (data) fields. The main entry points (which must always be present) are the open, close, and strategy functions, as stored respectively in the b_open, b_close and b_strategy fields. The handling of each is described below (a dummy prefix of `xxxx_` being used for illustration).

Open

Function:

```
int xxxx_open (dev, flag)
  dev_t dev;
  int flag;
```

Purpose:

This call is used to initialise access to the device specified as *dev*; *flag* defines the intended mode of access. The call may occur for two reasons: (a) the system *mount* operation is being performed, and the device will be used as a filing system, or (b) a user process with suitable access rights to the device is opening it for block-mode access (it is not possible for the driver to distinguish these cases). In either case, subsequent operations on the device will be performed through the strategy function. The call is made on behalf of, and in *foreground* context of, a particular process.

Hence it is valid to use the sleep/wakeup mechanism if operations required to validate the open may require a delay (typically awaiting completion on an interrupt-driven basis). Note that block device drivers should only call sleep with a high scheduling priority, that is, the second *prio* parameter to sleep() should have a value < PZERO; PRIBIO is the normal value. (Scheduling priorities, which are quite distinct from cpu priority levels, are defined in `param.h`.) In this way, signals to the process will not cause unwanted interruption of the operation.

The function is normally called at cpu priority level 0, usually referred to as spl (short for *system level priority*) 0. However, it may raise the priority to a higher level (PRIO_BIO, set by a call on the function splbio(), is standard) to prevent device interrupts during sensitive activities, or for other reasons of atomicity. The spl must be restored to its entry level before the function returns; this is normally done by the sequence:

```
int s;
...
s = splbio();
/* sensitive code here */
...
splx (s);
return (result);
```

Parameters:

dev is the complete device number. If the device supports multiple units and/or partitions, the minor number should be checked for validity; for example not all sub-units may be present, in which case the error code ENXIO should be returned if an attempt is made to access an absent unit.

The *flag* parameter will have one or both of the FREAD and FWRITE bits (defined in `file.h`) set in it, according to the desired access mode. Many drivers simply ignore this parameter; however a driver for a device which is inherently read-only, or which can be physically write-protected, can if required, refuse the open (returning EROFS) if the FWRITE bit is set.

Return:

The open function indicates success or failure to its caller by means of the value returned. This should be 0 if the open was successful, or a standard error code (macros beginning with `E`, defined in `errno.h`) on failure. Note that the older mechanism of error indication used in other versions of UNIX, by setting the global variable u.u_error, is not used in this context. If an I/O error occurs during any initialisation activity, EIO may be returned. EBUSY is the error code used to indicate that the device could not be opened because it was already in use; however there is rarely cause to implement this type of restriction for typical block devices – any number of open calls are normally allowed. In fact it is common for a block device open function to be called many times before the corresponding close. For example, the *fsck* program normally opens the device it is checking twice – once for read operations and once for write operations. It is therefore unusual, and generally unwise, to restrict the number of open calls allowed.

Strategy

Function:

```
xxxx_strategy (bp)
  struct buf *bp;
```

Purpose:

This function is the principal active component of a block device driver's interface. All requests to the driver for I/O activity are channelled through this function. Refer to the system include file `buf.h` for fuller details of the buffer structure; however, in brief, this structure defines an area of memory (starting at `bp->b_un.b_addr` and extending for `bp->b_bcount` bytes) as the source or destination for respectively a write or read operation (as determined by `bp->b_flags &` B_READ) on the device whose device number is specified as `bp->b_dev`, starting at block `bp->b_blkno` of the device. (Note that `bp->b_blkno` is always measured in terms of a standard device block-size, defined as DEV_BSIZE, 512 bytes.) The strategy call is the means of instructing the driver to perform the given read or write operation; however the driver is not assumed to have completed the operation when the strategy call returns – instead, a standard routine `biodone(bp)` should be called by the driver when the operation on the buffer has been completed (or abandoned because of error). This can happen at any stage, typically when processing a *completion* interrupt from the device.

An arbitrary number of operations may be pending at one time – for example the kernel may go through a list of delayed I/O transfers, starting each in turn by calling the strategy function, with no attempt to wait for completion of each before processing the next. Hence the driver is required to record the buffer in a set of uncompleted operations. For this purpose, the fields `bp->av_forw` and `bp->av_back` (both of type `(struct buf *)`) are normally used as links for a doubly-linked list: it is common (at least on disc drives) for the driver to sort these in some way for improved performance.

Unlike the open and close calls, the strategy function is NOT called on behalf of a particular user process; therefore the sleep mechanism must not be used during the strategy call. The cpu priority at the point of call is not fixed, although it will never be higher than PRIO_BIO, so the call `splbio()` should be used to protect sensitive operations like buffer queue manipulation against possible interference from device interrupts. An implication of this last point is that the priority level set for hardware interrupts from block devices must not exceed PRIO_BIO.

When the transfer described in the buffer structure has been processed, the call `biodone(bp)` informs the kernel of operation completion, after which point the driver should not attempt to access the buffer in any way.

Normally, the operation will succeed, in which case at the point of calling biodone the B_ERROR bit in `bp->flags` should be clear and `bp->b_error` be 0 to indicate success, and bp->b_resid should also be set to 0 showing that no bytes of the transfer remain unprocessed. If however an unrecoverable error does occur (it is normal for drivers to attempt a number of retries if this is sensible), the flag B_ERROR should be or'ed into `bp->b_flags`, and the field `bp->b_error` should be set to an error code (commonly EIO) to indicate this, before biodone is called. If the operation failed completely, `bp->b_resid` should equal `bp->b_bcount`; otherwise (if some of the data was transferred) `bp->b_resid` should contain a count of the number of bytes not transferred; `bp->b_bcount` should always contain the same value as at the call of `xxxx_strategy`.

The strategy function will normally perform a consistency check on the values of the buffer's fields: normally the only way in which a request may be in error is if it overflows the end of storage space on the logical device `bp->b_dev` (where this may identify a partition of a larger physical device). A check for this would be that `bp->b_blkno`, plus the number of blocks implied by `bp->b_bcount` (transfer byte count) goes beyond the end of the last block of the logical device. One situation in which this error may occur is when the kernel generates an automatic read-ahead request for a subsequent block, following a user request to read the last block of the logical device, i.e. the request starts exactly at the end of the device (assuming the device size is an exact multiple of the file-system block size, normally 4Kb or 8Kb). For convenience of kernel treatment it is conventional for block drivers to handle this case specially, by not setting the B_ERROR flag or the `bp->b_error` field, but rather by setting `bp->b_resid` to equal `bp->b_bcount` (indicating that no data was transferred). In any case of the request being unacceptable to the driver, including this one, biodone should be called before return from the strategy function itself; thus the request is viewed as having been processed immediately.

Parameter:

bp is a pointer to a buffer structure being passed into the control of the block device driver – the kernel will not attempt to modify any part of the buffer structure or associated memory area after the call on xxxx_strategy, until the driver relinquishes control of the buffer and passes it back to the kernel using the call biodone(bp).

Return:

Although the strategy function is implicitly defined as an int function, this merely reflects C's treatment of functions without explicit result types – no value is returned, and the caller never expects one. As described above, errors are indicated by setting the `b_flags` and `b_error` fields of the buffer structure at the point where `biodone(bp)` is called, which may be a considerable time after the return from the strategy function.

Close

Function:

```
int xxxx_close (dev, flag)
  dev_t dev;
  int flag;
```

Purpose:

This call is used to indicate to the driver that access to the device is now complete (no file handles or mount points are active on it). Many drivers do nothing with this call; however drivers for demountable devices should ensure that all pending activity on the device has terminated before the call returns (normally it is only possible for write operations to remain outstanding at this point). As with the open call, `sleep()` may be used if it is necessary for the process to wait for a particular condition.

Parameters:

These are as for the open call. The *flags* field will contain the logical bitwise *or* of all the *flags* values passed to the open function since the device was opened, ie, if any open since the last close included the FWRITE bit, then that bit will be set in the *flags* parameter to the close function, and similarly for the FREAD bit.

Return:

For historical reasons the close function is normally assumed to succeed – ie, the returned value is in most cases ignored. However for completeness it should return 0, in case the value is ever interpreted in future versions.

Subsidiary entry points

There are two further functions which are normally included in the bdevsw interface structure – these are the *size* and *dump* functions (called via the `b_size` and `b_dump` function pointer fields).

The purpose of the size function is to return to the kernel the size, in units of BDEV_SIZE (512) bytes, of the particular device identified by the device number passed as parameter. For a partitioned device this would be the size of the particular partition specified via the minor device number. In this way the kernel can determine, for instance, the size of swap partitions, and can limit user operations on block devices to lie within the device bounds. The declaration of this function should have the form:

```
int xxxx_size (dev)
 dev_t dev;
```

The dump function entry point is a carry-over from an earlier implementation of BSD UNIX, in which it may be used when the kernel has crashed with a panic. Its purpose is to dump the machine's physical memory onto a block device (as a direct contiguous transfer operation, rather than treating the device as a file-system). It is not used in the current version of RISC iX; nevertheless the entry point should be provided for compatibility with possible future inclusion of this feature. For now, a block device driver should simply return ENXIO for this function call; the code therefore looks like:

```
int xxxx_dump (dev)
  dev_t dev;
{
    return ENXIO;
}
```

Data fields in bdevsw entrys

There are three further fields to consider; `d_flags`, `d_name` and `d_drive_shift`; each of these holds a data value related to the driver, rather than the address of an item in the driver module itself.

The `d_flags` field is a general purpose field intended to contain flag bits describing particular characteristics of the device. Currently only one flag bit is tested by the system, this is the flag B_TAPE, which indicates that the device cannot be treated as a completely random access device. In particular a device with this flag set will automatically be mounted *read-only* if it is ever mounted as a file system; note that this flag does not mean that the device is guaranteed to be a real tape drive, although if real tape drivers provide a block device interface, they will normally set this bit. Apart from this bit no other use is currently made of the `d_flags` field, which will therefore commonly be left as 0.

The `d_name` field contains a device identifier consisting of two letters. This is used in the system boot sequence to allow a boot program to specify which devices should be used for the root file system device, the initial swap device and the panic dump device (although as described in Section 3.4 the dump mechanism is currently inoperative). As the kernel is started up, the boot program passes in to it names and sub-unit codes for the device to be used for each purpose: the kernel boot code uses the identifier to find the actual block device required, by scanning the `bdevsw` table `d_name` entries looking for the specified name. The two-character name must be

unique to the driver, and typically resembles the driver prefix in some way, but this latter point is not enforced. It is merely necessary to document the name used for a given driver so that the boot program can be configured to use it as required.

The `d_drive_shift` field is also used in the kernel boot sequence – it allows the boot program to specify not only the (major) device by name, but also the drive and partition on that device, to be used for a given purpose, without requiring all block devices to interpret their minor device number in the same way. What is assumed is that if a device driver breaks its minor device number up into unit (or drive) and partition fields, then (a) this is done on the basis of a whole number of bits for each (ie each component can have 2**K values) and (b) the partition bits will be the lowest N bits of the minor device number and the drive bits the remaining 8-N. For a given driver, the `d_drive_shift` field records the value of N, ie the number of bits used for the partition field, and hence the number of bits by which a specified drive number should be shifted to occupy its correct position within a constructed minor device number.

Raw interface to a block device driver

It is common for drivers for block device drivers to provide also a character or *raw* interface to the device, whereby a process with suitable access rights for the particular entry in the directory `/dev` can make transfers directly between the device and its own memory, without going through the normal kernel buffer mechanisms. The advantages of doing this are that certain types of operations, notably backup of a disc partition, can be executed more efficiently than if they were forced to use kernel buffering. In particular the size of transfers can be much greater than that conventionally used for user-programmed access to a block device (as distinct from file system access) – in the current version of RISC iX this is done in 2Kb units, whereas many types of storage device readily permit transfers of much larger amounts in one operation. This raw access scheme is supported in RISC iX by means of a standard set of functions which take care of the main tasks involved in setting up such a transfer, and cause the transfer to be performed through the normal strategy mechanism. The amount of additional coding required in a typical block device to support raw access is therefore quite small. The method of interface is to provide not only an entry in the `bdevsw` table, but also an entry in the `cdevsw` table, by which means a user's `read(2)` and `write(2)` system calls are passed right through to the device driver via the `xxxx_read` and `xxxx_write` entry points.

Besides the `read` and `write` entry points, the `cdevsw` structure also includes `open` and `close` functions, as for the block device; these are called on user open and close calls, in a fashion similar to that for the block device. Note that block and character device open operations are counted quite separately, so if the existing block device functions are used for this purpose, it is possible for the close function to be called twice with no intervening open, once for the block interface and once for the raw interface (in either order). If the close function performs no useful work, this will obviously not be a problem; however as intimated above in the description of the close function, physically demountable device drivers in particular need to keep track of opens and closes in order to ensure consistency across possible changes of the media (they should attempt to prevent, or at least complain about, media changes whilst a device is open). In this latter case it is possible to cope with combined block and raw access by providing distinct open and close functions for each access class – each function can then do suitable accounting for whether an open of its own type is active.

It is possible (although not frequently needful) to allow further driver-specific control functions by providing an `xxxx_ioctl` entry point; one case where this is useful is with block-addressable tape drives, which normally also provide a raw interface with standard `ioctl` commands as defined in the system manual page `mtio(4)`.

One final entry point in the `cdevsw` structure which is relevant in the context of raw device I/O is the `d_sectorsize` field. For a block device driver providing a raw interface also, an `xxxx_sectorsize` function should be defined, returning the size in bytes of the smallest sector transfer acceptable to the device. Typically, DEV_BSIZE (512 bytes) is the minimum value normally returned, even if the device's physical sectors are smaller than 512 bytes, and most devices have physical sectors no larger than 2Kb. The reason this value is determined via a function call rather than, say, having a fixed integer value stored in the field is that different sub-devices of a major device may have different minimum sector sizes. By this means it is possible for the system to limit access requests to be in units of this minimum sector size. (Note: this mechanism is not used in initial versions of RISC iX).

Sample outline block device drivers

Here is the code for an imaginary device driver, controlling some sort of hard disc drive by means of an expansion card. The driver is written to allow a number of such cards to be fitted, and provides independent access to each drive.

First a header file:

```
/* file: iecd.h */

/*
 * Header for Imaginary Expansion Card - Disc
 *
 * Copywright 1989, Imaginary Devices Ltd.
 *
 * The real one follows...
 *
 * Copyright (C) 1989, Acorn Computers Ltd.
 *
 */

/* Description of the imaginary expansion card disc (iecd) */

/* physical drive organisation */

#define D_SECTORS       56      /* device sectors per track */
#define D_HEADS         4       /* = tracks/cylinder */
#define D_CYLS          2310    /* total cylinders/drive */
#define D_SECSIZE       256     /* note: small sectors */

/* Define the ratio of standard device block size to our
 * physical sector size.
 */
#define SECS_PER_BLK    (DEV_BSIZE/D_SECSIZE)

/*
 * Number of standard-sized (512-byte) blocks per track - note that
 * there is an exact number of these, so no fiddling around with
 * odd 256-byte sectors is needed.
 */
#define BLKS_PER_TRACK  ((D_SECTORS*D_SECSIZE)/DEV_BSIZE)
/* and per complete cylinder */
#define BLKS_PER_CYL    (BLKS_PER_TRACK*D_HEADS)

/*
 * Definition of the registers on the imaginary expansion card disc
 * controller.  Note that because of the way ARM and the expansion
 * card bus hardware work we have to fiddle this a bit.  When we write
 * a 16-bit quantity to the controller, we actually write a whole word
 * (since ARM does not support half-word operations on memory), and
 * the data must be placed in the top half of the word written (the
 * low 16 bits are ignored). When reading from a 16-bit register we
```

```
 * must get the data out of the low 16-bits of the word we read.
 */
struct iecdregs
{
    /*
     * The first word in the register block is dual-purpose: it is the
     * the control register if we are writing to it, or the status
     * register when we read from it.  For simplicity (although we
     * might think of using a union) we just use a macro to get the
     * two names.
     */
    unsigned int c_control;
#define c_status c_control
    /*
     * The next register is for sending commands to the controller, and
     * is write-only.
     */
    unsigned int c_command;
    /*
     * Then follows the param register (used for setting in such things
     * as cylinder number, head number, sector number/count, etc).  Again
     * this is write-only.
     */
    unsigned int c_param;
    /*
     * Finally the data register - this is bidirectional - we write to
     * it data to be put onto the disc, and read from it data we have
     * requested from the disc.  It is also used when the controller is
     * telling us about error cases.
     */
    unsigned int c_data;
};

/*
 * The following gives the address of the disc controller registers
 * in expansion card memory space.  REGS_LOC is the byte offset of the
 * base of the register set, within the normal 8Kb expansion card
 * address space for a given I/O cycle-speed and slot-number.
 * The iecd card is designed to be accessed with FAST I/O cycle
 * timings.
 */
#define REGS_LOC        0x800
#define IECD_REGS(slot) \
    ((struct iecdregs *)(XCB_ADDRESS(FAST,slot) + REGS_LOC))

/*
 * Define macros to handle the 16-bit shifts
 * typical use would be -
 *
 *      write_reg (regs->c_command, C_WRITE);
 *
 *      status = read_reg(regs->c_status);
 */
#define write_reg(reg, value)   reg = (value) << 16
#define read_reg(reg)           ((reg) & 0xFFFF)

/*
 * Definitions of commands, status etc. Much simplified from
 * the average real disc controller, and much more convenient!
 */

/* Bits in the control register */
#define CON_TWO_BUFF    (1 << 0) /* use internal double buffering */
#define CON_ECC         (1 << 1) /* do automatic ECC */
#define CON_FASTSEEK    (1 << 2) /* if set, use high-speed seek */
#define CON_HEADSWITCH  (1 << 3) /* do multi-head operations */
#define CON_AUTOSTEP    (1 << 4) /* do multi-cylinder ops */

/*
 * Command values written to the command reg (after all
 * parameters have been sent via the param reg).
 */
#define CMD_RESET       0x00    /* clears controller state */
#define CMD_READ        0x22    /* read sectors */
#define CMD_WRITE       0x24    /* write sectors */
```

```
#define CMD_SEEK        0x35    /* seek to cylinder */
#define CMD_TEST        0x41    /* controller internal test */

/* Bits in the status register */
#define STAT_BUSY       (1 << 0) /* executing a command */
#define STAT_CMDERR     (1 << 1) /* bad command */
#define STAT_PARMERR    (1 << 2) /* bad parameter to command */
#define STAT_DRVERR     (1 << 3) /* error from drive */
#define STAT_DATA       (1 << 4) /* ready for data transfer */
#define STAT_DONE       (1 << 5) /* command completed OK */

/* EOF iecd.h */
```

Then the main body of code:

```
/* file: iecd.c */

/*
 * Driver for Imaginary Expansion Card - Disc
 *
 * Copywright 1989, Imaginary Devices Ltd.
 *
 * The real one follows...
 *
 * Copyright (C) 1989, Acorn Computers Ltd.
 *
 */


/*
 * The following set of #includes is typical for a block
 * device driver.  The cc command used when compiling
 * this module should include the relevant directories in
 * which these various files are found, by means of the
 * -I<dir> flag.
 */
#include "param.h"
#include "systm.h"
#include "buf.h"
#include "conf.h"
#include "dir.h"
#include "file.h"
#include "user.h"
#include "kernel.h"
#include "vnode.h"
#include "vm.h"
#include "uio.h"

/* error logging stuff */
#include "syslog.h"

/* busy-wait delay support */
#include "delay.h"

/* includes defining the expansion bus and interrupt interfaces */
#include "int_hndlr.h"
#include "xcb.h"

/* finally the include file defining our own device */
#include "iecd.h"

/*
 * Maximum number of drives (1 per card) we will handle -
 * there are a total of 4 expansion card slots, but we
 * limit ourselves to 2 cards for reasonableness - it is
 * pointless to waste space when we are unlikely ever to
 * run more than that number.
 */
#define MAX_DRIVE       2

/*
 * The following contains the actual number of drives present,
 * up to a limit of MAX_DRIVE - this is determined at boot
 * time in the expansion card initialisation sequence.
```

```
 * Drives are always numbered in the range 0..(n_drive-1),
 * in the order their controller cards were found in the XCB
 * manager's scan of the expansion card bus.
 */
static int n_drive;

/* macros for handling minor device number */

#define DRIVENO(mindev) ((mindev) >> 3)
#define PARTNO(mindev)  ((mindev) & 7)
#define MAX_PART        8


static struct drive
{
    /* flags for access to the drive */
    int d_flags;
    /* values of d_flags bits */
#define DF_OPENED       (1 << 0)
#define DF_OPENING      (1 << 1) /* open sequence interlocks */
#define DF_OPENWAIT     (1 << 2)
    int d_open_status;
    /*
     * partition table for a drive - this is read in on first
     * open of any partition of that drive.
     */
    struct part
    {
        int p_start;            /* first cylinder of partition */
        int p_cyls;             /* how many cylinders in partition */
        int p_rdonly;           /* partition access flag */
    } d_part[MAX_PART];
    struct iecdregs *d_regs;    /* address of drive controller regs */
    unsigned char d_slot;       /* physical expansion card slot */
    unsigned char d_cmd;        /* operation drive is currently doing */
    long *d_mem;                /* memory address for data transfer */
    int d_sectors;              /* how many sectors remain to be done */
    int d_cur_cyl;              /* current drive head position */
    int d_next_cyl;             /* target drive head position (for seek) */
    int d_last_cyl;             /* head position after a data transfer */
    int d_retries;              /* counts error recovery attempts */
    struct buf d_ioq;           /* header record for I/O operations queue */
} drive_info[MAX_DRIVE];

#define MAX_RETRIES     5       /* number of attempts on an operation */

/*
 * Various forward references for static (i.e. local) functions
 * used before they are defined.
 */
static int check_drive();
static int open_drive();
static void start_drive();
static void drive_int();


/* Expansion Card Bus manager interface code first */

/*
 * Interrupt handler records, one for each possible source of
 * IRQs, i.e. for each iecd card which may be configured.
 */
static struct int_hndlr card_ih[MAX_DRIVE];


void iecd_init_hi (slot)
  int slot;
{
    int drive, status;
    struct drive *di;

    /* An iecd card has been found in the specified slot */
    if (n_drive == MAX_DRIVE)
    {
        /* We have already initialised as many cards as we cope with */
```

```
        printf ("ignoring iecd card in slot %d\n", slot);
        return;
    }
    /* set up our structures for the drive */
    drive = n_drive++;          /* allocate drive number */
    di = &drive_info[drive];    /* address drive record */
    di->d_slot = slot;          /* record physical slot number */
    di->d_regs = IECD_REGS(slot); /* calculate card regs address */
    /*
     * Now do some basic hardware checks on the card/drive interface:
     * these all happen in foreground (we are not allowed to use the
     * sleep/wakeup mechanism at this stage - for delays we just loop);
     * also, interrupts are presently disabled.
     */
    if ((status = check_drive (drive)) != 0)
    {
        /* The hardware isn't working properly */
        printf ("iecd%d, slot %d: card failed hardware test, status=%d\n",
                drive, slot, status);
        return;
    }
    /* OK, a working drive - report configuration on the console */
    printf ("iecd%d: slot %d\n", drive, slot);

    /*
     * Finally we must declare our card's interrupt source to the
     * interrupt system, via the expansion card bus manager.  To
     * do this we initialise the ih_fn and ih_farg fields of a
     * static int_hndlr structure, and pass its address along with
     * the physical slot number and the desired IRQ priority to
     * decl_xcb_interrupt().
     */
    card_ih[drive].ih_fn = drive_int;
    /* set drive number as arg to be passed to the interrupt handler */
    card_ih[drive].ih_farg = drive;
    /*
     * Ask XCB manager to arrange that the IRQ has normal block I/O
     * device priority.
     */
    decl_xcb_interrupt (slot, &card_ih[drive], PRIO_BIO);
}

iecd_init_lo (slot, irqs)
  int slot, irqs;
{
    /*
     * This is the low-priority initialisation routine - as it happens
     * we don't need to do anything here, but we might have chosen to
     * do things like read partition tables etc off the drive at this
     * point, rather than as part of the device open sequence.  Note
     * that the system is not yet running in full multi-processing
     * context and use of the sleep/wakeup mechanism is therefore
     * impossible, and fatal if tried!
     *
     * As documented in "xcb.h", this function will be called twice
     * for each slot containing an iecd device, once with XCB
     * interrupts disabled (marked by irqs == 0), and the second time
     * with them enabled (irqs == 1).  On the second call, therefore,
     * we could do some work in conjunction with our interrupt
     * handler, waiting in a polling loop in the main code, if this
     * were desired.
     */
}


/*
 * iecd_shutdown arranges that the iecd card in the given slot
 * is in a stable, non-interrupting state, as if it had been
 * given a hardware reset.  This is very easy.  Note that we can
 * be called even for cards which have failed their hardware test
 * on initialisation... beware.
 */
iecd_shutdown (slot)
  int slot;
{
```

```
    /* The following should be sufficient in our case... */
    struct iecdregs *regs = IECD_REGS(slot);
    /*
     * The controller reset command clears everything to
     * a suitable state.
     */
    write_reg (regs->c_command, CMD_RESET);
}


/* Main block device interface follows */

int iecd_open (dev, flag)
  dev_t dev;
  int flag;
{
    int mindev = minor(dev);
    int drive = DRIVENO(mindev);
    struct drive *di;
    struct part *pt;

    /* check for valid drive number */
    if (drive >= n_drive)
        return ENXIO;

    /* address drive information record */
    di = &drive_info[drive];

    /* check for first open of this drive */
    if ((di->d_flags & DF_OPENED) == 0)
    {
        /*
         * No open has yet completed, but interlock on possible
         * parallel open attempt, since only one attempt ought
         * to be made at once, and it is possible that some process
         * is sleeping in the middle of open_drive();
         */
        int s = splbio ();      /* for protection (slight paranoia) */
        if (di->d_flags & DF_OPENING)
        {
            /* someone else trying now - let them do the work */
            do
            {
                di->d_flags |= DF_OPENWAIT;
                sleep ((caddr_t)di, PRIBIO);
            } while (di->d_flags & DF_OPENING);
            /* they have finished: di->d_open_status now set up */
        }
        else
        {
            int  sleepers;
            /* mark that we are doing an open of this drive */
            di->d_flags |= DF_OPENING;
            /*
             * Initialise the drive, read in and validate partition
             * table, etc; this is performed by a separate function.
             * During execution of this code we may sleep waiting for
             * completion.
             */
            di->d_open_status = open_drive (drive);
            /* mark that an open attempt has been made */
            di->d_flags |= DF_OPENED;
            /* check whether any other process is waiting for us */
            sleepers = (di->d_flags & DF_OPENWAIT) != 0;
            /* clear out the interlock flags */
            di->d_flags &= ~(DF_OPENING|DF_OPENWAIT);
            /* wake up anyone who was waiting */
            if (sleepers)
                wakeup ((caddr_t)di);
        }
        splx (s);
    }

    /* check whether drive has been successfully opened */
    if (di->d_open_status != 0)
```

```
        return di->d_open_status; /* no - give up */

    /* address the relevant partition description */
    pt = &drive_info[drive].d_part[PARTNO(mindev)];

    /* a partition of size 0 is undefined and inaccessible */
    if (pt->p_cyls == 0)
        return ENXIO;

    /* check for read-only flag on partition */
    if ((flag & FWRITE) && pt->p_rdonly)
        return EROFS;

    /* all seems to be in order */
    return 0;
}

int iecd_close (dev, flag)
  dev_t dev;
  int flag;
{
    /* We don't need to do anything special here */
    return 0;
}


/*
 * iecd_strategy - where I/O requests are processed.
 */
iecd_strategy (bp)
  struct buf *bp;
{
    int mindev = minor(bp->b_dev);
    int drive = DRIVENO(mindev);
    struct drive *di;
    struct part *pt;
    int nblks, s;

    /* Set up for the specific drive and partition involved */
    di = &drive_info[drive];
    pt = &di->d_part[PARTNO(mindev)];

    /*
     * We permit block size-multiple transfers only, starting on
     * a word boundary in memory.
     */
    if (((unsigned int)bp->b_bcount % DEV_BSIZE) != 0 ||
        ((long)bp->b_un.b_addr & (sizeof(int)-1)) != 0)
    {
        bp->b_flags |= B_ERROR; /* flag an error */
        bp->b_error = EINVAL;   /* set the error code in */
        bp->b_resid = bp->b_bcount; /* no data moved */
        biodone (bp);           /* pass buffer back to kernel control */
    }

    /* work out size of partition in system device block units */
    nblks = pt->p_cyls * BLKS_PER_CYL;

    /* check for transfer outside partition bounds */
    if (bp->b_blkno + (bp->b_bcount / DEV_BSIZE) > nblks)
    {
        /*
         * don't complain too hard if exactly at end of partition
         * - this helps read-ahead handling.
         */
        if (bp->b_blkno != nblks)
        {
            /* quite unacceptable request */
            bp->b_flags |= B_ERROR;
            bp->b_error = ENXIO;        /* set the error code in */
        }
        bp->b_resid = bp->b_bcount; /* no data moved */
        biodone (bp);           /* pass buffer back to kernel control */
    }
    /*
```

```
     * Everything seems OK - now queue and possibly start the transfer.
     *
     * First we ensure that queue manipulation is not messed up by
     * interrupts from the controller
     */
    s = splbio();

    di->d_ioq.b_qcnt++;         /* one more item in queue now */

    bp->av_forw = NULL;         /* clear forward link, for queuing */

    if (di->d_ioq.b_actf == NULL)
    {
        /* There was nothing happening: install buffer on empty Q */
        di->d_ioq.b_actf = bp;  /* sits at head of queue */
        di->d_ioq.b_actl = bp;  /* and also at tail */
        /*
         * Start up the operation, since the controller must be
         * idle; first set retries on operation to 0.
         */
        di->d_retries = 0;
        start_drive (drive);
    }
    else
    {
        /*
         * There is already at least one entry on the queue, which
         * will be being processed at the moment.  For simplicity, we
         * just place this request at the end of the current queue
         * (first come first served ordering).  It might be useful at
         * some stage to attempt some sorting on the queue entries.
         */
        di->d_ioq.b_actl->av_forw = bp;
        bp->av_back = di->d_ioq.b_actl;
        di->d_ioq.b_actl = bp;
    }
    splx (s);                   /* restore SPL */
}


/* iecd_size returns the size of the specified device */
int iecd_size (dev)
  dev_t dev;
{
    int mindev = minor(dev);
    /*
     * Get the size from the appropriate partition of the
     * appropriate drive, these being determined from the
     * minor device number.  We return the size in units
     * of DEV_BSIZE: the macro BLKS_PER_CYL gives the number
     * of these per cylinder on our disc.
     */
    return (drive_info[DRIVENO(mindev)].d_part[PARTNO(mindev)].p_cyls *
            BLKS_PER_CYL);
}


/* iecd_dump is the normal trivial implementation for now */
int iecd_dump (dev)
  dev_t dev;
{
    return ENXIO;
}


static void command_drive (di, bp)
  struct drive *di;
  struct buf *bp;
{
    int mindev = minor(bp->b_dev);
    struct iecdregs *regs = di->d_regs;
    unsigned int start_block = (bp->b_blkno +
                                (di->d_part[PARTNO(mindev)].p_start *
                                 BLKS_PER_CYL));
    unsigned int start_cyl = start_block / BLKS_PER_CYL;
```

```
/*
 * We must be located on the right cylinder before we start a read
 * or write operation, however access to subsequent cylinders is
 * handled by the controller, since we have set the AUTOSTEP bit
 * in its control register...
 */
if (di->d_cur_cyl != start_cyl)
{
    /*
     * We need to do a seek operation.  Tell it where to go - the
     * SEEK command takes one 16-bit parameter, which is the
     * target cylinder number (starting at 0) + 1.  Disc drive
     * controller designer's brains seem to have odd glitches
     * sometimes....
     */
    write_reg (regs->c_param, start_cyl + 1);
    write_reg (regs->c_command, CMD_SEEK); /* off she goes.. */
    /* remember what we are up to, for the next interrupt */
    di->d_cmd = CMD_SEEK;
    /* and where we are going to... */
    di->d_next_cyl = start_cyl;
}
else
{
    /*
     * We're in the right place, set up for the data transfer.
     * For impenetrable reasons, we have to remind the
     * controller which cylinder it's on, besides specifying
     * the head and sector to start with.  Thanks to the
     * HEADSWITCH and AUTOSTEP functions (aren't Imaginary devices
     * wonderful!) which we always configure in the controller,
     * we don't have much else to worry about....
     */
    unsigned int cblock = start_block % BLKS_PER_CYL; /* block no in cylinder */
    unsigned int track = cblock / BLKS_PER_TRACK; /* which track to start on */
    /* compute start sector in physical device units */
    unsigned int sector = (cblock % BLKS_PER_TRACK) * SECS_PER_BLK;
    unsigned int blocks;
    /*
     * Note that all the parameters to the controller
     * are offset by one, as for seek.
     */
    write_reg (regs->c_param, start_cyl+1);
    /*
     * The head (track) and start sector get combined in the second
     * 16-bit parameter.
     */
    write_reg (regs->c_param, (track+1) << 12 | (sector + 1));

    /*
     * Third and final parameter is the sector count, which we
     * compute from the transfer byte count.  We record this and
     * the next memory address in the drive structure, for use
     * on data traffic interrupts.
     */

    /* The byte count is already known to be a block-size multiple */
    blocks = bp->b_bcount / DEV_BSIZE;
    di->d_sectors = blocks * SECS_PER_BLK;

    /* And the memory address to be word aligned... */
    di->d_mem = (int *)bp->b_un.b_addr;

    /* Set the last parameter in place */
    write_reg (regs->c_param, di->d_sectors);

    /*
     * Now start it off by giving it the appropriate command,
     * which we record for testing on the next interrupt.
     */
    di->d_cmd = (bp->b_flags & B_READ) ? CMD_READ : CMD_WRITE;
    write_reg (regs->c_command, di->d_cmd);

    /*
```

```
         * Compute which cylinder the last block of the transfer
         * will be transferred to/from, so we can track where we
         * are.
         */
        di->d_last_cyl = (start_block + blocks - 1) / BLKS_PER_CYL;
    }
}

static void start_drive (drive)
  int drive;
{
    struct drive *di = &drive_info[drive];
    struct buf *bp;
    /* see if there is anything on the queue to be processed */
    if ((bp = di->d_ioq.b_actf) == NULL)
        return;                 /* nothing to do */
    command_drive (di, bp);
}

/*
 * drive_int is the function called when an iecd drive
 * controller card interrupts.  We have arranged (in
 * iecd_init_hi()) that the argument passed in is the
 * logical drive number concerned.
 */
static void drive_int (drive)
  int drive;
{
    struct drive *di = &drive_info[drive];
    struct iecdregs *regs = di->d_regs;
    struct buf *bp;
    int status;

    /*
     * Get the head of queue, which is the transfer we are
     * processing at the moment.
     */

    if ((bp = di->d_ioq.b_actf) == NULL)
    {
        /* eh? nothing going on - must be a spurious interrupt */
        log (LOG_NOTICE, "iecd%d: spurious int\n", drive);
        return;                                 /* just ignore it */
    }

    /* first get drive status */
    status = read_reg (regs->c_status);

    /* check for errors */
    if (status & (STAT_DRVERR))
    {
        /*
         * Here we read the error code from the controller
         * and handle it appropriately.  If possible, we
         * retry the operation, unless the retry count has
         * reached its limit, when we log the error and
         * abandon the operation.
         */
        int code = read_reg (regs->c_data);
        if (++di->d_retries >= MAX_RETRIES)
        {
            log ("iecd%d: drive error %04x\n", drive, code);

            /* move on to the next item in the queue */
            di->d_ioq.b_actf = bp->av_forw;
            di->d_retries = 0;  /* for next operation */
            di->d_ioq.b_qcnt--; /* one less item in Q */

            /* flag the error */
            bp->b_flags |= B_ERROR;
            bp->b_error = EIO;
#ifndef CALC_RESID
            /*
             * For simplicity, ignore what data may have been
             * successfully transferred (it might be tricky to work
```

```
             * out the right numbers) and say that nothing was moved.
             * This is normally quite acceptable.
             */
            bp->b_resid = bp->b_bcount;
#else   CALC_RESID
            /*
             * Attempt to compute how much data was transferred.  We
             * assume that for a read operation, everything the drive
             * actually fed us was valid data, hence the residual
             * count is just the data in the sectors not yet moved
             * (held in the drive record).  For a write, we assume
             * here that the error meant that the last sector we sent
             * to the drive was NOT successfully written to the disc
             * (since the controller always has to get a complete
             * physical sector from us before it can attempt to write
             * it to the disc) so the sectors count will be down by
             * one.  It is possible for some error cases not to fit
             * this pattern, but we won't attempt to cope with those
             * for the moment.
             */
            if (di->d_cmd == CMD_WRITE)
                bp->b_resid = (di->d_sectors + 1) * D_SECSIZE;
            else if (di->d_cmd == CMD_READ)
                bp->b_resid = di->d_sectors * D_SECSIZE;
            else
                /*
                 * An error on the initial seek operation, we assume.
                 * No data will have moved at all; note that d_sectors
                 * will not have been computed, so we just use the
                 * original byte count.
                 */
                bp->b_resid = bp->b_bcount;
#endif CALC_RESID
            /* operations on this buffer now abandoned */
            biodone (bp);
            /* go do next item on queue, if any */
            start_drive (drive);
            return;
        }
        else
        {
            /*
             * Retry it - just calling command_drive is enough,
             * but first we set the "current cylinder" flag to
             * an impossible value, to force a re-seek attempt.
             */
            di->d_cur_cyl = -1; /* invalid */
            command_drive (di, bp);
            return;
        }
    }
    /* OK, no problem: now what was going on? */
    if (di->d_cmd == CMD_SEEK)
    {
       /* got where we were going OK */
       di->d_cur_cyl = di->d_next_cyl;
       /* start data transfer operation */
       command_drive (di, bp);
    }
    else
    {
       /* must be read/write - check for data traffic */
       if (status == (STAT_BUSY|STAT_DATA))
       {
            /*
             * It wants a (256 byte) sector to be moved (in or
             * out): check which way we're going on this operation.
             *
             * The following code MIGHT go faster if we wrote it
             * in assembler, but there's not much in it.  If our
             * device allowed data transfers in several (half-)word
             * chunks at a time, via sequential I/O space addresses,
             * we would definitely want to go to assembler, in
             * order to use ARM's ldm/stm instructions for maximum
             * speed.
```

```
         *
         * First set up pointers for one sector in memory.
         * The d_mem field of the drive record tells us where to
         * start.  We also compute where we will end up, for a
         * convenient loop-termination test.
         */
        register int *data = di->d_mem;
        register int *limit = data + (D_SECSIZE/sizeof(int));
        /*
         * The following coding explicitly handles the 16 bit
         * shifts normally treated by read/write_reg().
         */
        if (di->d_cmd == CMD_READ)
        {
            do
            {
                /*
                 * Read in 32 bits per loop (as two half words).
                 * Note: it is vital that this be done as two
                 * separate C statements rather than as e.g.
                 * *data++ = (regs->c_data & 0xFFFF) |
                 *           regs->c_data << 16;
                 * since C implies nothing about the order in
                 * which the two register reads are done, or
                 * even whether the register is read twice at all!
                 * With a very clever compiler, this is still
                 * a difficulty, even using two statements as below.
                 * Later versions of the Acorn (NorCroft) ARM C
                 * compiler in pcc mode allow the ANSI-style
                 * "volatile" construct, which is usefully applied
                 * in such situations.  Alternatively the use of
                 * assembly code avoids these problems.
                 */
                register int d;
                /*
                 * First get low 16-bits (half-word) and mask the
                 * undefined top half of the data.
                 */
                d = regs->c_data & 0xFFFF;
                /*
                 * Then combine this with the high half-word shifted
                 * into place, and store it out.
                 */
                *data++ = d | regs->c_data << 16;
            } while (data < limit);
        }
        else
        {
            /* must be a write op */
            do
            {
                int word = *data++;
                /*
                 * First write low half, shifted up as required
                 * for I/O space write operations.
                 */
                regs->c_data = word << 16;
                /*
                 * Then write high half, which is in the right
                 * place in the word already.
                 */
                regs->c_data = word;
            } while (data < limit);
        }
        /* update the memory pointer */
        di->d_mem = limit;
        --di->d_sectors;    /* count down how many remain */
    }
    else if (status == STAT_DONE)
    {
        /*
         * Command complete: move on to the next item
         * in the queue.  First record where we have
         * now reached in terms of cylinders.
         */
```

```
            di->d_cur_cyl = di->d_last_cyl;

            di->d_ioq.b_actf = bp->av_forw;
            di->d_ioq.b_qcnt--; /* one less item in Q */

            di->d_retries = 0;  /* for next operation */

            /* operations on this buffer now finished */
            biodone (bp);

            /* start processing next item on queue, if any */
            start_drive (drive);
        }
    }
}


static int check_drive (drive)
  int drive;
{
    struct drive *di = &drive_info[drive];
    struct iecdregs *regs = di->d_regs;
    int conreg, status;

    /* reset the drive to start with */
    write_reg (regs->c_command, CMD_RESET);

    /*
     * Check that the reset has completed OK; wait a little
     * while first (should finish inside 10uSec normally).
     */
    MICRODELAY(50);                             /* wait at least 50 uSec */

    if (read_reg (regs->c_status) & STAT_BUSY)
        /* oh dear - it should have finished - give up */
        return 1;                               /* diagnostic code */

    /*
     * Now check the controller's internals - this should take no
     * more than 200 uSec.
     */
    write_reg (regs->c_command, CMD_TEST);

    MICRODELAY(1000);                           /* wait a good while */

    if (read_reg (regs->c_status) & STAT_BUSY)
        /* oh dear - it didn't finish - give up */
        return 2;                               /* diagnostic code */

    /* read the command status (1 byte) back from the data register */
    status = read_reg (regs->c_data) & 0xFF;

    if (status != 0)
        /* return distinctive diagnostic, including the code we got */
        return status + 1000;

    /* all seems OK - go ahead now */

    /*
     * Configure the control register - we use as much of the
     * controller's "intelligence" as we can, to simplify our
     * own task.
     */
    conreg = CON_TWO_BUFF | CON_ECC | CON_HEADSWITCH | CON_AUTOSTEP;

    /*
     * If the appropriate link is set on the board (as determined
     * by reading part of the card's ID space), the attached drive
     * will permit high-speed seeks, so we use that facility of the
     * controller.
     */
    if (XCB_ID(di->d_slot)->data[0].b & 1)
        conreg |= CON_FASTSEEK;

    /* set the control register up */
```

```
    write_reg (regs->c_control, conreg);

    /* all done */
    return 0;
}


static int open_drive (drive)
  int drive;
{
    /*
     * Code here should arrange to read and validate the partition
     * table stored at a fixed position on the drive, and do any
     * one-time setting up of the drive.
     */
}


/* The remaining code handles raw I/O */

/*
 * Allocate a private buffer to be used to implement raw I/O transfer
 * requests.  We pass this to physio() which sets it up and passes it
 * through to iecd_strategy().  Note that although we may have multiple
 * raw I/O requests in effect at once (from different processes, and
 * possibly for different drives) this is so rare that we only bother
 * declaring a single buffer: physio() looks after the interlocks to
 * prevent multiple use of it, so there is no problem.  If we expected
 * to have a lot of raw I/O activity, we might want to declare multiple
 * buffers (e.g. one per drive, or on a first-come-first-served basis)
 * but typically there is little advantage in doing this.
 */
static struct buf raw_io_buf;

/*
 * iocheck ensures that raw I/O transfer requests satisfy
 * certain constraints imposed by the hardware & software.
 */
static int iocheck (uio)
  struct uio *uio;
{
    struct iovec *iov;
    int i;
    /*
     * Raw I/O transfer requests must start on a 512-byte boundary on
     * the logical disc, since physio() does its computations of block
     * number on this basis.  The uio_offset field of a uio structure
     * gives the byte position that the user process has got to on the
     * associated file-descriptor (by means of previous read, write or
     * lseek calls) - we use this as the byte offset from the start of
     * the partition.
     */
    if ((uio->uio_offset & (DEV_BSIZE-1)) != 0)
        return EINVAL;
    /*
     * Now, for each segment of the transfer (there may be more than
     * one, if a process uses the "writev" system call), check that it
     * is a multiple of 512 bytes in size (since we can only request
     * whole sectors from the disc) and starts in memory on a whole
     * word (integer, 4 bytes) boundary, since this is all that our
     * transfer code will cope with,
     */
    iov = uio->uio_iov;
    for (i = uio->uio_iovcnt; i > 0; --i, ++iov)
        if ((iov->iov_len & (DEV_BSIZE-1)) != 0 ||
            ((int)iov->iov_base & ((sizeof(int))-1)) != 0)
            return EINVAL;
    /* No segment failed the test - we can proceed */
    return 0;
}


/*
 * iecd_read processes read requests coming through the
 * raw interface.  Most of the work is done for us by
```

```
 * physio().
 */
int iecd_read (dev, uio)
  dev_t dev;
  struct uio *uio;
{
    int  err;
    if ((err = iocheck (uio)) != 0)
        return err;
    /*
     * physio does all the main work for us, including organising the
     * user virtual memory to be read into or written from, ensuring
     * that it is all resident and contiguous in memory.  It also
     * breaks up large requests into pieces of a size determined by
     * the routine we pass to it (we use the standard "minphys", and
     * don't worry about the details).  It calls the strategy routine
     * we pass to it with the address of the buffer which it has set
     * up with the details of the transfer or each part of it.  Physio
     * waits for each part of the transfer to complete (or an error
     * to occur) before it starts the next part or returns.  It returns
     * a status value (0 -> success, else error code), which we simply
     * pass back to our caller.
     */
    return physio (iecd_strategy, &raw_io_buf, dev, B_READ, minphys, uio);
}


/*
 * iecd_write is exactly like iecd_read with the exception of the
 * direction of transfer.
 */
int iecd_write (dev, uio)
  dev_t dev;
  struct uio *uio;
{
    int  err;
    if ((err = iocheck (uio)) != 0)
        return err;
    return physio (iecd_strategy, &raw_io_buf, dev, B_WRITE, minphys, uio);
}


int iecd_sectorsize (dev)
  dev_t dev;
{
    /* For simplicity, we use DEV_BSIZE as our minimum sector size */
    return DEV_BSIZE;
}

/* EOF iecd.c */
```

5.4 Terminal driver structure

Interactive terminals, commonly referred to as 'ttys', form a significant subset of the general class of character devices, and the structure of a `cdevsw` record reflects this. RISC iX, like other versions of the UNIX operating system, provides a substantial amount of generic code to handle the interface to a standard serial line terminal, so the work of a tty driver writer is largely to do with handling the specific features of the hardware. Notably, practically all the work of the main tasks, of interpreting the sequence of characters received from the hardware and post-processing bytes being transmitted, is performed by generic code in the kernel rather than in the driver. The sample outline terminal driver in the following section should act as a general pattern for the structure of a typical tty driver - all the main points of interfacing are illustrated.

Sample outline terminal drivers

Again, this driver has been invented for an imaginary device - there is no hardware corresponding to this code! - but it is related to a real device driver (that for the 65C51 used in the R140's built-in serial interface). Again, like the example block device driver, it is interfaced through the standard expansion card scheme so the

XCB support facilities of RISC iX are used. The driver is relatively complex (it copes with multiple hardware lines on multiple cards) but the main source code file - reasonably well commented - occupies only about 21 Kbytes, thus showing the benefits of the generic tty handling mechanisms of RISC iX (and BSD UNIX in general).

First the expansion card specific header:

```
/* iecs.h
 *
 * Header for Imaginary Expansion Card - Serial
 *
 * Copywright, 1989 Imaginary Devices Ltd.
 *
 * Copyright (C) 1989, Acorn Computers Ltd.
 *
 */

/*
 * Header file defining the main programming interface of the IECS
 * card.  Some aspects of the individual ACIA chip (Asynchronous
 * Communications Interface Adaptor) interface are common with the
 * R140 internal serial port, so we use the file "acia.h" in the
 * system device serial devices directory "sys/dev/serial" for those
 * parts.
 */

/* Number of 65C51 ACIAs on each card */
#define LINES_PER_CARD  2

/*
 * Address of the ACIA register set (organised address-wise as for
 * the R140 internal 65C51) for each line.  This is relative to the
 * card slot start address in XCB I/O space - our card is operated
 * with medium-speed I/O cycles.
 */

#define CARD_ACIA(slot, line) \
    ((ACIARef)(XCB_ADDRESS(MEDIUM,slot) + (0x0800+0x200*(line))))

/*
 * Address of the IRQ register per card - this read/write byte-wide
 * register contains bits indicating the presence of IRQs for each
 * line (when read) and when written, the top bit controls the master
 * IRQ enable state - 0 means the card will not generate IRQs, 1
 * allows it to.  The card IRQ line is the OR of all the individual
 * ACIA IRQ lines.  Writing to this register only affects the IRQ
 * enable bit, not the individual IRQ flags, so read-modify write
 * coding is unnecessary.
 */

#define CARD_IRQ(slot) \
    ((unsigned char *)(XCB_ADDRESS(MEDIUM,slot) + 0x4C0))

/* Define the interrupt enable bit */
#define IR_IENABLE      (1 << 7)

/* EOF iecs.h */
```

Then the actual driver code:

```
/*
 * iecs.c
 *
 * Driver for Imaginary Expansion Card - Serial
 *
 * Copywright, 1989 Imaginary Devices Ltd.
 *
 * Copyright (C) 1989, Acorn Computers Ltd.
 *
 */

#include "param.h"
#include "systm.h"
#include "ioctl.h"
#include "tty.h"
#include "dir.h"
#include "user.h"
#include "proc.h"
#include "map.h"
#include "buf.h"
#include "vm.h"
#include "conf.h"
#include "bkmac.h"
#include "file.h"
#include "uio.h"
#include "kernel.h"
#include "syslog.h"
#include "ioc.h"

#include "xcb.h"
#include "int_hndlr.h"

/*
 * We use (some) defs from Acorn's "es" (internal R140
 * device) driver for the 65C51 chip, so include the
 * published header.
 */
#include "../serial/acia.h"

/* and some more definitions of our own */
#include "iecs.h"

/* Maximum number of cards we will support at once */
#define MAX_CARD        2

/* macros for handling minor device numbers */
/* card number */
#define CARDNO(mindev)  ((unsigned int)(mindev) / LINES_PER_CARD)
/* line number (card-relative) */
#define LINENO(mindev)  ((unsigned int)(mindev) % LINES_PER_CARD)

/* Defaults */
#define ISPEED  B9600                   /* initial line speed */
#define IFLAGS  (EVENP|ODDP|ECHO)       /* initial line flags */

/* Derived values */
#define MAX_LINES       (MAX_CARD*LINES_PER_CARD)

#define ON      1                       /* turn acia on */
#define OFF     0                       /* turn it off */


/*
 * The following public data item groups the tty structures for all
 * possible lines in a contiguous block, as assumed by the default
 * select function for ttys, ttselect().
 */

struct tty iecs_tty[MAX_LINES];         /* tty data structures */

struct line
{
    ACIARef         l_acia;             /* -> chip registers */
```

```
    unsigned char  l_tx_off;            /* flag for transmitter state */
    unsigned char  l_cur_dcd;           /* reflects modem status */
    unsigned char  l_cur_dsr;           /* ditto */
    unsigned char  l_open;              /* flag for line open */
};

static struct card
{
    struct line    c_line[LINES_PER_CARD];
    unsigned char  c_slot;              /* physical slot for card */
    /*
     * The next field is a pointer to the card interrupt
     * control register - it is R/W, showing which acia(s)
     * is/are interrupting at any point, and controlling
     * the master card IRQ line.
     */
    unsigned char  *c_irq;
} card_info[MAX_CARD];

static int n_card;                      /* #cards found */
static int n_line;                      /* #lines in total */


/* Forward refs */
int  iecs_start ();
static void switch_modem ();
static void  card_int ();

/* Interrupt handler structure, one per card */
static struct int_hndlr  card_ih[MAX_CARD];


iecs_init_hi (slot)
  int slot;
{
    /*
     * The XCB manager has found an iecs card in slot "slot".
     * Let's initialise it, unless we've already been told about
     * MAX_CARD cards.
     */
    int card, i, status;
    struct card *cd;
    card = n_card;
    if (card == MAX_CARD)
    {
        printf ("ignoring iecs card in slot %d\n", slot);
        return;
    }
    /* Report configuration on the console */
    printf ("iecs[%d-%d]: slot %d\n",
            card*LINES_PER_CARD, card*(LINES_PER_CARD+1)-1,
            slot);
    /* Count up known cards */
    ++n_card;
    n_line += LINES_PER_CARD;           /* and available lines */

    /* Address the record for this card */
    cd = &card_info[card];

    cd->c_slot = slot;

    /* Set up card interrupt register pointer */
    cd->c_irq = CARD_IRQ(slot);
    for (i = 0; i < LINES_PER_CARD; ++i)
    {
        struct line *line = &cd->c_line[i];
        ACIARef acia = CARD_ACIA(slot, i);
        line->l_acia = acia;            /* record regs pointer */
        /* Reset the chip */
        reset_acia (acia);
        /* Read its status */
        status = acia->status;
        /*
         * Get the current state of the DCD and DSR lines of the
         * hardware interface.  Note the chip's negative logic.
```

```
         */
        line->l_cur_dcd = !(status & ACIA_STAT_DCD);
        line->l_cur_dsr = !(status & ACIA_STAT_DSR);
        /* Reset leaves the transmitter disabled - track this */
        line->l_tx_off = 1;
    }
    /* Declare our interrupt handler for this card */
    card_ih[card].ih_fn = card_int;
    card_ih[card].ih_farg = card;
    decl_xcb_interrupt (slot, &card_ih[slot], PRIO_TTY);
    /*
     * Finally, enable interrupts from the card.  Note that as
     * explained in iecs.h, we can just write this bit - the
     * individual irq indicator lines are not affected.
     */
    *cd->c_irq = IR_IENABLE;
}

/*
 * Note: there is no low-priority initialisation routine; the
 * init_lo field of our xcbinfo record in xcbconf.c is 0.
 */


/* Card shutdown function */
iecs_shutdown (slot)
  int slot;
{
    /*
     * Since this function will be called for ALL iecs cards in
     * the machine, we can't assume that the full initialisation
     * sequence has been done for all slots (we have ignored all
     * cards above MAX_CARD).  Hence to address a card's interrupt
     * control register, the simplest way is just to use the
     * hardware addressing macros direct, and not bother with
     * card_info[] etc.
     */
    int lno;
    /* Resetting the card is easy: reset each acia. */
    for (lno = 0; lno < LINES_PER_CARD; ++lno)
    {
        ACIARef acia = CARD_ACIA(slot,lno);
        reset_acia (acia);
    }
    /*
     * Then ensure it cannot interrupt.  Writing 0 to the irq control
     * register clears the master interrupt enable.
     */
    *CARD_IRQ(slot) = 0;
}


iecs_open (dev, flag)
  dev_t dev;
  int flag;
{
    struct tty *tp;
    int mindev;
    struct line *lp;
    int oldpri;

    /* Check we have the requested device installed */
    mindev = minor (dev);
    if (mindev >= n_line)
        return (ENXIO);

    /* Address the relevant tty and hardware line control structures */
    tp = &iecs_tty[mindev];
    lp = &card_info[CARDNO(mindev)].c_line[LINENO(mindev)];

    /* Sensitive operations ahead: keep interrupts out */
    oldpri = spltty ();

    /* Set up default output start function */
    tp->t_oproc = iecs_start;
```

```
    if ((tp->t_state & TS_ISOPEN) == 0)
    {
        /*
         * First open of this device since boot or last close.
         *
         * Set up the default special characters.
         */
        ttychars (tp);
        /* Set other defaults */
        tp->t_state |= TS_HUPCLS;
        tp->t_ispeed = ISPEED;
        tp->t_ospeed = ISPEED;
        tp->t_flags = IFLAGS;
        /* Reflect software state into hardware */
        iecs_param (dev);
    }
    else if ((tp->t_state & TS_XCLUDE) && u.u_uid != 0)
    {
        /*
         * Exclusive-access bit set, and our caller is not
         * the super-user - reject this open.
         */
        splx (oldpri);
        return (EBUSY);
    }
    /* Turn on any attached modem */
    switch_modem (dev, ON);
    /*
     * Test whether we have carrier as yet.  This may be
     * a local line, in which DCD will be wired to ON.
     */
    if (lp->l_cur_dcd)
    {
        /* yep - all OK */
        tp->t_state |= TS_CARR_ON;
    }
    else
    {
        /* Go wait till carrier is present */
        tp->t_state &= ~TS_CARR_ON;
        do
        {
            tp->t_state |= TS_WOPEN;
            /* It is conventional to sleep on the raw Q for this */
            sleep ((caddr_t) &tp->t_rawq, TTIPRI);
        } while ((tp->t_state & TS_CARR_ON) == 0);
    }
    splx (oldpri);
    /* Return via the current line-discipline's open routine */
    return ((*linesw[tp->t_line].l_open) (dev, tp));
}


iecs_close (dev, flag)
  dev_t dev;
  int   flag;
{
    struct tty *tp;
    int oldpri;
    /* Address relevant tty structure */
    tp = &iecs_tty[minor(dev)];
    /* Call the line discipline close routine */
    (*linesw[tp->t_line].l_close) (tp);
    /*
     * Go to suitable priority to block interrupts while
     * adjusting hardware state.
     */
    oldpri = spltty ();
    /* Ensure any line break condition is cleared */
    iecs_ioctl (dev, TIOCCBRK, NULL, 0);
    /*
     * If we were stuck waiting for open, or the HUPCLS flag is set,
     * turn off ("hang up") the modem line.
     */
    if ((tp->t_state & (TS_HUPCLS|TS_WOPEN)) || (tp->t_state & TS_ISOPEN) == 0)
```

```
        switch_modem (dev, OFF);
    /* Finished tweaking hardware bits */
    splx (oldpri);
    /* Finally use the standard close-down code to tidy up */
    ttyclose (tp);
}


iecs_read (dev, uio)
  dev_t dev;
  struct uio *uio;
{
    /*
     * Process user read() request - this is trivial since the
     * line discipline code does all the work - all we have to
     * do is identify the tty from the device number, and call it.
     */
    struct tty *tp = &iecs_tty[minor(dev)];
    return ((*linesw[tp->t_line].l_read) (tp, uio));
}


iecs_write (dev, uio)
  dev_t dev;
  struct uio *uio;
{
    /* User write() request is as easy as read(). */
    struct tty *tp = &iecs_tty[minor(dev)];
    return ((*linesw[tp->t_line].l_write) (tp, uio));
}


/*
 * ioctl call - we support a few standardised serial line
 * hardware controls, otherwise the generic code handles
 * most things.
 */
iecs_ioctl (dev, cmd, data, flag)
  dev_t dev;
  int cmd;
  caddr_t data;
  int flag;
{
    /* Identify tty structure concerned */
    int mindev = minor(dev);
    struct tty *tp = &iecs_tty[mindev];
    int status, s;
    ACIARef acia;
    /*
     * Try a sequence of subsidiary ioctl function calls first.  The
     * convention is for all such functions (but not this top-level one)
     * to return -1 for unrecognised ioctl commands, otherwise a status
     * code (0 or error) if they have handled it or faulted it.
     * First we try the current line discipline's ioctl entry point.
     */
    status = (*linesw[tp->t_line].l_ioctl) (tp, cmd, data, flag);
    if (status >= 0)
        return (status);
    /* It didn't know it - try the standard ttioctl routine next.. */
    status = ttioctl (tp, cmd, data, flag);
    if (status >= 0)
    {
        /*
         * One of the set ttioctl handles; some of these commands
         * involve changes to the software state which should be
         * reflected in the hardware.
         */
        if (cmd == TIOCSETP || cmd == TIOCSETN ||
            cmd == TIOCLBIS || cmd == TIOCLBIC || cmd == TIOCLSET)
            iecs_param (dev);
        return (status);
    }
    /* addres the hardware for potential changes */
    acia = card_info[CARDNO(mindev)].c_line[LINENO(mindev)].l_acia;
    /*
```

```
     * If the command is one we process here, the hardware will
     * be modified, so raise SPL first in case it is.
     */
    s = spltty ();
    /* set default result */
    status = 0;
    switch (cmd)
    {
      case TIOCSBRK:
        /* Set BRK now (disables RTS) - worry about RTS etc. later */
        acia->command |= ACIA_TIC_NTX_RTS_BRK;
        break;

      case TIOCCBRK:
        /* Clear BRK: RTS is derived from DTR */
        cmd = acia->command & ~ACIA_CMD_TIC;
        if (cmd & ACIA_CMD_DTR)
            cmd |= ACIA_TIC_TX_RTS;
        acia->command = cmd;
        break;

      case TIOCSDTR:
        /* Sets DTR+RTS; clears any BRK */
        acia->command = ((acia->command & ~(ACIA_CMD_TIC))
                         | ACIA_TIC_TX_RTS | ACIA_CMD_DTR);
        break;

      case TIOCCDTR:
        /* Clears DTR+RTS; clears any BRK */
        acia->command &= ACIA_CMD_TIC | ACIA_CMD_DTR;
        break;

        /* To Be Done: TIOCMOD{G,S} (also TIOCM{SET,BIS,BIC,GET}?) */

      default:
        /* The command has still not been identified, we return ENOTTY. */
        status = ENOTTY;
        break;
    }
    splx (s);
    return (status);
}


iecs_param (dev)
{
    int mindev;
    struct tty  *tp;
    ACIARef acia;
    int s;
    unsigned commandreg, controlreg;
    static unsigned char acia_speed[1+EXTB] =
    {
        0,                                          /* unused */
        ACIA_CON_RCS | ACIA_SPEED_50,
        ACIA_CON_RCS | ACIA_SPEED_75,
        ACIA_CON_RCS | ACIA_SPEED_110,
        ACIA_CON_RCS | ACIA_SPEED_134,
        ACIA_CON_RCS | ACIA_SPEED_150,
        0xFF,                                       /* 200 ignored */
        ACIA_CON_RCS | ACIA_SPEED_300,
        ACIA_CON_RCS | ACIA_SPEED_600,
        ACIA_CON_RCS | ACIA_SPEED_1200,
        ACIA_CON_RCS | ACIA_SPEED_1800,
        ACIA_CON_RCS | ACIA_SPEED_2400,
        ACIA_CON_RCS | ACIA_SPEED_4800,
        ACIA_CON_RCS | ACIA_SPEED_9600,
        ACIA_CON_RCS | ACIA_SPEED_19200,            /* EXTA = 19200 */
        0xFF                                        /* EXTB ignored */
    };
    s = spltty ();
    mindev = minor(dev);
    tp = &iecs_tty[mindev];
    acia = card_info[CARDNO(mindev)].c_line[LINENO(mindev)].l_acia;
    if (tp->t_ispeed == 0)
```

```
    {   /* hang up modem */
        switch_modem (dev, OFF);
        splx (s);
        return;
    }
    if ((controlreg = acia_speed[tp->t_ispeed]) == 0xFF)
        /* new speed not valid - ignore and use current setting */
        controlreg = acia->control & (ACIA_CON_RCS | ACIA_CON_SPEED);
    if (tp->t_ispeed == B110)
        /* For 110 Baud, we set up to use 2 stop bits */
        controlreg |= ACIA_CON_SBN;
    commandreg = acia->command & ~(ACIA_CMD_PMC|ACIA_CMD_PME);
    /*
     * The following treatment of the LITOUT and PASS8 flags is
     * somewhat dubious.  LITOUT being set affects not only the
     * output processing, as specified, but also input parity,
     * given the limitations of the 65C51's parity handling;
     * similarly PASS8 (an input control flag) causes parity
     * generation on output to be turned off.  This is a mess
     * which could be fixed by extra code elsewhere in this
     * driver (by doing software parity checks) however several
     * of the VAX BSD4.3 serial line drivers behave exactly the
     * same way, without obvious serious problems, so we leave
     * it for now...
     */
    if (tp->t_flags & (RAW|LITOUT|PASS8))
    {
        controlreg |= ACIA_WL_8;
        commandreg |= ACIA_PMC_SPACE;           /* but PME off, so no parity */
    }
    else
    {
        controlreg |= ACIA_WL_7;
        commandreg |= ACIA_CMD_PME;             /* send parity bit.. */
        /* decide parity polarity */
        commandreg |= (tp->t_flags & EVENP) ? ACIA_PMC_EVEN : ACIA_PMC_ODD;
    }
    /* Finally, install the settings into the hardware */
    acia->command = commandreg;
    acia->control = controlreg;
    splx (s);
}


/* Interrupt handler */

static void tx_int (), rx_int(), modem_int ();

void  card_int (card)
  int card;
{
    struct card *cd = &card_info[card];
    int irq = *(cd->c_irq);             /* get int req register */
    int lno;
    /* Local data and status buffering */
    unsigned char l_data  [LINES_PER_CARD];
    unsigned char l_status[LINES_PER_CARD];

    /*
     * To minimise delays and thereby reduce the risk of
     * character overrun, we process the interrupting lines
     * in two phases: first, read the status and any input
     * character from each interrupint acia.  Then, go do the
     * appropriate thing for each line according to the
     * status we read.
     */
    for (lno = 0; lno < LINES_PER_CARD; ++lno)
    {
        /*
         * Check the card interrupt register, which shows
         * the interrupt status of each line chip in the
         * corresponding bit position (0..LINES_PER_CARD-1).
         */
        if (irq & (1 << lno))
        {
```

```
            /*
             * Interrupt from Line lno is present.
             * Address the relevant acia chip.
             */
            ACIARef acia = cd->c_line[lno].l_acia;
            /* Read the chip's status and record it */
            int status = acia->status;
            l_status[lno] = status;

            /*
             * Allow for spurious interrupts (rare).
             */
            if ((status & ACIA_STAT_IRQ) == 0)
                /* ghost IRQ:  remove this bit from the irq set */
                irq &= ~(1 << lno);
            else
                /* Check for RX data ready and pick it up if so */
                if (status & ACIA_STAT_RDRF)
                    l_data[lno] = acia->data;
        }
    }
    /* Now go round again, having done the most urgent tasks */
    for (lno = 0; lno < LINES_PER_CARD; ++lno)
    {
        if (irq & (1 << lno))
        {
            struct line *line = &cd->c_line[lno];
            /*
             * Work backwards from normal to produce a minor
             * device number from the card and line numbers.
             */
            int mindev = card * LINES_PER_CARD + lno;

            /* Hence determine the tty structure */
            struct tty *tp = &iecs_tty[mindev];

            /* Get this line's recorded status */
            int status = l_status[lno];

            /*
             * Compute up-to-date DCD/DSR state.
             * Note negative logic.
             */
            int chip_dcd = !(status & ACIA_STAT_DCD);
            int chip_dsr = !(status & ACIA_STAT_DSR);

            if (chip_dcd != line->l_cur_dcd ||
                chip_dsr != line->l_cur_dsr)
            {
                /* Change in DCD/DSR - go handle */
                modem_int (tp, line, chip_dcd, chip_dsr);
            }

            /* Check for receiver interrupt.. */
            if (status & ACIA_STAT_RDRF)
                rx_int (tp, status, l_data[lno]);

            /* Then transmitter.. */
            if (status & ACIA_STAT_TDRE)
                tx_int (tp, status);
        }
    }
}


/* Handle Receiver interrupt */
static void rx_int (tp, status, data)
  struct tty *tp;
  int status, data;
{
    /*
     * Wake up anyone who might happen to have been waiting
     * for the device to open.
     */
    if ((tp->t_state & TS_ISOPEN) == 0)
```

```
    {
        wakeup ((caddr_t)&tp->t_rawq);
        if ((tp->t_state & TS_WOPEN) == 0)
            return;
    }

    /* Check for errors in the received data */
    if (status & ACIA_STAT_FE)
        /*
         * Convert frame errors (caused by line break or speed mismatch)
         * to NUL (if line is in raw mode) or the current interrupt
         * character.
         */
        data = (tp->t_flags & RAW) ? 0 : tp->t_intrc;

    /*
     * Data loss is marked by (status & ACIA_STAT_OVRN) - however there isn't
     * much we can do about it - we might care to log it but this is likely
     * to make matters worse (especially on 9600 baud lines or faster) since
     * log() can take a significant time to execute.
     * The following code is left for perusal, but not compiled: note -
     *
     *           NEVER DEFINE "notdef"!!!!
     *
     * because the "#ifdef notdef" construct is commonly used for such
     * purposes as below, and cases where code is incomplete or not
     * functional.
     */

#ifdef notdef
    if (status & ACIA_STAT_OVRN)
    {
        /*
         * At least one character (we don't know exactly how many)
         * has been lost on the wire because we weren't able to
         * respond to an interrupt quickly enough.
         *
         * In the following, we have to extract the minor device
         * number since we were only passed a tty pointer.
         */
        log (LOG_WARNING, "iecs%d: char overrun\n", minor(tp->t_dev));
    }
#endif notdef

    if (status & ACIA_STAT_PE)
        /* Parity error - decide how to treat it */
        if (((tp->t_flags & (EVENP|ODDP)) == EVENP) ||
            ((tp->t_flags & (EVENP|ODDP)) == ODDP))
            /* ignoring bad parity chars */
            return;

    /*
     * Pass the character to upper levels via the line
     * discipline's input routine.
     */
    (*linesw[tp->t_line].l_rint) (data, tp);
}


/*
 * tx_int is called when the transmitter has completed sending
 * one character and is ready to take the next.
 */
static void tx_int (tp)
  struct tty *tp;
{
    tp->t_state &= ~TS_BUSY;            /* no longer transmitting */
    if (tp->t_line)
        (*linesw[tp->t_line].l_start) (tp);
    else
        iecs_start (tp);
}


/*
```

```
 * modem_int handles changes in the state of the DCD and DSR
 * interface lines to a modem.  Note that at the moment we
 * don't do anything with the DSR bit, however, other than keep
 * track of its state.  This routine is called at spltty.
 */
static void modem_int (tp, lp, new_dcd, new_dsr)
  struct tty *tp;
  struct line *lp;
  int new_dcd, new_dsr;
{
    if (new_dcd != lp->l_cur_dcd)
    {
        /* Carrier transition */
        if (new_dcd)
        {
            /* carrier now on */
            if ((tp->t_state & TS_CARR_ON) == 0)
                (void) (*linesw[tp->t_line].l_modem) (tp, 1);
        }
        else
        {
            /* carrier gone away */
            if (tp->t_state & TS_CARR_ON)
                if ((*linesw[tp->t_line].l_modem) (tp, 0) == 0)
                    switch_modem (tp->t_dev, OFF);
        }
    }
    lp->l_cur_dcd = new_dcd;
    lp->l_cur_dsr = new_dsr;
}


/*
 * iecs_start is the routine called (from the line discipline code
 * and other places) to attempt to transmit a character if this
 * is possible.  Note that it is called from both foreground and
 * interrupt states, at indeterminate cpu priority, so we must
 * use spltty() to bar interrupts, and be very careful in what
 * we do.  Most of this code follows a standard structure (yep,
 * including the gotos!) so be careful if you think of doing
 * things much differently.
 */
extern int ttrstrt ();
iecs_start (tp)
  struct tty *tp;
{
    /*
     * All that we get passed is a tty pointer, so to get at our
     * own private information about a line, we must first extract
     * the minor device number from tp->t_dev;
     */
    int mindev = minor (tp->t_dev);
    struct line *line = &card_info[CARDNO(mindev)].c_line[LINENO(mindev)];
    ACIARef acia = line->l_acia;
    int s;
    s = spltty ();
    if (tp->t_state & TS_BUSY)
        goto  outb;
    if (tp->t_state & (TS_TIMEOUT|TS_TTSTOP))
        goto outn;
    if (tp->t_outq.c_cc <= TTLOWAT(tp))
    {
        if (tp->t_state & TS_ASLEEP)
        {
            tp->t_state &= ~TS_ASLEEP;
            wakeup ((caddr_t)&tp->t_outq);
        }
        if (tp->t_wsel)
        {
            selwakeup (tp->t_wsel, tp->t_state & TS_WCOLL);
            tp->t_wsel = 0;
            tp->t_state &= ~TS_WCOLL;
        }
    }
    if (tp->t_outq.c_cc == 0)
```

```
        goto outn;
    if ((tp->t_flags & (RAW|LITOUT)) == 0)
    {
        int cc = ndqb (&tp->t_outq, 0200);
        if (cc == 0)
        {
            /* Delay info pending */
            cc = getc (&tp->t_outq);
            timeout (ttrstrt, (caddr_t)tp, (cc&0x7f) + 6);
            tp->t_state |= TS_TIMEOUT;
            goto outn;
        }
    }
    if (line->l_tx_off)
    {
        acia->command = (acia->command & (~ACIA_CMD_TIC)) | ACIA_TIC_TX_RTS;
        line->l_tx_off = 0;
    }
    acia->data = getc (&tp->t_outq);
    tp->t_state |= TS_BUSY;
outb:
    splx (s);
    return;
outn:
    line->l_tx_off = 1;
    acia->command = (acia->command & ~ACIA_CMD_TIC) | ACIA_TIC_NTX_RTS;
    splx (s);
}


/*
 * Turn the modem on or off.  Note: we are called at spltty,
 * so no extra protection on hardware access is required.
 */
static void switch_modem (dev, flag)
  dev_t dev;
  int flag;
{
    int mindev = minor (dev);
    ACIARef acia = card_info[CARDNO(mindev)].c_line[LINENO(mindev)].l_acia;
    if (flag == OFF)
        acia->command &= ~(ACIA_CMD_DTR|ACIA_CMD_TIC); /* stop any BRK too */
    else
        /* Set DTR+RTS */
        acia->command = ((acia->command & ~ACIA_CMD_TIC)
                         | ACIA_TIC_TX_RTS | ACIA_CMD_DTR);
}


/* EOF iecs.c */
```

5.5 Expansion card bus driver interface

As explained earlier, the standard UNIX device driver interface requires the driver to provide a number of entry points, which are compiled into the *bdevsw* and/or *cdevsw* tables as required; these are used for the purpose of interfacing with the kernel proper and include the `open`, `close`, `read`, `write`, `ioctl` and `strategy` functions (depending on device class). For devices interfaced via the expansion card bus, further entry points should be defined to handle initialisation and shutdown of the card. A separate file xcbconf.c contains a table of these entry points along with other information, for devices which may be present on the expansion card bus. The file `/usr/include/arm/xcb.h` contains definitions for the expansion card bus interface; the many comments in this file are intended to help in understanding the structure of the XCB device interface – it is important to read it before attempting to write an XCB device driver.

5.6 Device driver components

A device driver, as supplied for distribution, should consist of the following software and documentation components:

- An object module (or modules) containing the code and data structures for the driver; this will be linked with modules for the rest of the kernel and other device drivers, to produce a complete kernel image file.
- A fragment of C source code, giving the names of the main entry points to the driver; this will be included in the file `conf.c` (cdevsw and/or bdevsw tables) for each driver to be included in a kernel build.
- (For expansion card devices) a fragment of C source code defining the XCB device interface, including device identification information and the names of entry points for device control, to be included in the kernel configuration file `xcbconf.c`. Note that the published system header file `xcb.h` contains #define lines for all the expansion card types known at release time, including manufacturer and individual card ID codes.
- (Optional) a header file to be included in the directory `/usr/include/dev`, defining any particular characteristics and functionality (eg via `ioctl` calls), for programming access to the device.
- Documentation of the device name(s), type, minor device number interpretation and access permissions; this is to allow installation of the device in the `/dev` directory. NOTE: actual major device numbers for a device should not be given - the major device number for a device is determined by the position of its entry in the cdevsw or bdevsw tables, as arranged in `conf.c`, and this may vary according to individual kernel configuration.
- Documentation for access to the device from a program. This is conveniently organised as an nroff file suitable for inclusion in the manual page directory `/usr/man/man4` (which by convention contains documentation for various devices); see any of the existing pages for the conventions used in manual page layout.

Chapter 6 – Shared libraries

Shared libraries

6.1 Introduction

This document describes the RISC iX shared library scheme. It is assumed that the reader has an understanding of the C compiler in the UNIX environment and of the principles of separate compilation and link editing.

Reasons for using shared libraries

There are two basic reasons for wanting to use shared libraries:

- To reduce the memory occupancy of a running program by sharing code with other running programs.
- To reduce the disc occupancy of a set of programs by storing common code in a single place.

In addition to these aims the RISC iX shared library implementation tries to meet a number of other requirements:

a) Shared libraries should be of benefit to a wide variety of existing programs.

b) Use of shared libraries in the standard UNIX utilities should be beneficial.

c) Using a shared library in a particular program should require minimal changes to the program.

d) Converting an existing library to a shared library should require minimal changes to the code of the library.

e) The scheme should be applicable to a wide variety of existing libraries.

Tips for using shared libraries are given in section 6.3, Guidelines.

Some of these aims are in conflict – although a variety of shared library schemes exist all of these have some effect on the code which uses them. Requiring that changes be minimal to both library code and application code restricts the schemes which may be used.

The shared library mechanism in RISC iX was developed to help with the relatively restricted memory and disc space of the R140 machine, hence requirement (b) above. Although many UNIX utilities contain pieces of code of near identical functionality (for example the regular expression matching in `sed`, `ed`, `ex/vi/view`, `grep`, `egrep` and so on) this is frequently implemented using separate pieces of code – for example most of the programs which recognise regular expressions do not use the `regex(3)` library code to do so. To meet requirements (b) and (c) it was necessary to share existing library code. The only library which is common to the majority of UNIX utilities is `libc` – hence, by implication, the scheme must allow `libc` to be shared.

Basic principles of the RISC iX scheme

The RISC iX shared library scheme is extremely crude. A shared library is built by `ld(1)` in exactly the same manner as a normal program, however the special –Z flag causes the data to be linked at the top of the user virtual address space, rather than immediately after the program text.

When a program is linked with the shared library (or information derived from it by `mkshare(1)`) the linker does not include the text or data of the shared library into the resultant image. Instead it includes a reference to the library in the `a.out(5)` header. When the kernel loads the program it also obtains the code and data of the shared library; if the code is already loaded (because another program using the same shared library is already executing) it is shared.

This scheme has a number of characteristics:

- The code in a shared library looks exactly like the code in a non–shared library to the program; however the individual archive members of the original library are collected into a single unit, thus the program gets all of the code or none of it.
- The shared library is already linked – it can have no outstanding references to user code. Thus parts of the shared library cannot be replaced with alternative user program definitions.
- The data of the shared library is linked at the top of the user address space. Common symbols have already been defined, thus it is not possible to merge definitions with user program definitions. User programs which make (false) assumptions about the memory location of data may be surprised – the shared library data appears to be in the stack.
- The program must be run with the shared library with which it is linked. If the shared library is not available the kernel will refuse to execute the program.
- The shared library cannot be replaced with an alternative version in a linked program – in this scheme linking with a particular instance of a shared library is as permanent as linking with a non–shared version. Although the code of the library is not included in the program, the address of every entry point into that code and every piece of static data is bound into the program.

Shared libraries can share other libraries. Each program or library may only share one other library, thus shared libraries may be regarded as forming a hierarchical tree of dependency. (Normally the root of this tree is `libc`.). The data for a shared library is positioned immediately below the data for a library which it shares. The text of a shared library is positioned immediately above that of a library which it shares. When the program is linked its text is positioned immediately above that of the shared library which it shares and its data is positioned above that.

This tree structure has implications for the designer of a shared library. If two shared libraries are positioned on different branches in the tree they cannot be simultaneously shared by a single program. In general libraries of utilities should not be made into shared libraries, as a user program may wish to use them with an arbitrary mix of other shared libraries. The exception to this is `libc` – which is almost invariably the root library included in a program. The current RISC iX distribution includes the following shared libraries:

`libc.a`	–sharing nothing
`libX11.a`	–sharing `libc.a`
`libXt.a`	–sharing `libX11.a`

`LibX11.a` is the standard X11 interface library for C programs. Any X11 program will use it. `libXt.a` is a toolkit building library, any program using it must also use `libX11.a`. If other shared libraries are built the builder must decide whether it is conceivable that a program using the new library will also use X11. This may be sufficiently rare for it to be satisfactory to require the program to use a non-shared version of the new library or the X11 libraries. If the new library was an X11 toolkit – it should probably be built directly on top of `libX11.a`; it is most unlikely that a program would use two different toolkits.

Problems with using shared libraries

To execute a program which uses a shared library the kernel must not only be able to find and execute the program, it must also be able to find and execute all the shared libraries which it uses. To do this the libraries must be in the correct place with the correct attributes. Each library:

- Must have the same name as that stored in the relevant program (use `file(1)` or `phead(1)` to find the name).
- Must have the same time-stamp as that stored in the program. (`Phead(1)` will give the expected time-stamp from the program, and the actual time-stamp on the library. Mkshare will output the actual time-stamp value, which should correspond to the file name, if used in the form mkshare –nip / <library>.)
- Must have access rights which give execute (x) access to the user-id trying to execute the program.

If any of these are wrong the program will not execute. Because of this, and because most of the RISC iX system utilities share libc a system administrator must take particular care not to damage the libc shared library in /lib. Damaging this library will render your R140 totally unusable, because none of the programs which you need to put the damage right will execute. This is as effective a way of destroying your system as deleting `/etc/init` or `/bin/sh`.

In the event that you do rm or chmod the shared libc you will need to restore the damaged /lib/c:<time-stamp> file from a level 0 dump of the root partition. Notice that `/etc/restore` does not use a shared library. Alternatively you could boot the system from a floppy disc root partition (made earlier...) and copy the libc from that.

6.2 Shared library commands

The following commands are either affected by the implementation of shared libraries or have been changed (or implemented) to provide shared library support:

`ld`	Understands how to build shared libraries and how to link with them.
`mkshare`	A new program to produce a linkable file from a shared library. Libraries can be used directly, but they are unsuitable for inclusion in archives produced by the `ar(1)` command.
`phead`	A utility program which decodes the information in an `a.out(5)` header.
`cc`	`cc(1)` automatically links with *libc* – as `libc` is now shared any program linked via *cc* will be linked with at least that shared library.

These commands are discussed in more detail below. In addition to these commands several other commands now recognise and output appropriate information for shared libraries. These include `size(1)` and `file(1)`. Commands which use the

macros defined in `a.out(5)` will normally function correctly with shared libraries and related objects even though the magic numbers in the `a.out` header are different.

cc(1)

cc is frequently used as a front end for `ld`. Indeed many programmers do not realise that `cc` actually invokes `ld` to link the program. This can lead to considerable confusion. On a RISC iX system `cc` is actually the C compiler. On a standard UNIX system the real compiler is to be found in `/lib/ccom` and `cc` is only an argument decoder which invokes `/lib/ccp`, `/lib/ccom`, `as(1)` and `ld(1)` as appropriate. The RISC iX `cc(1)` command implements the functionality of both `cpp` and `ccom`, it does, however, still invoke *as* to deal with files whose name ends in `.s`, and `ld` to deal with the resultant object modules.

Like the standard UNIX `cc` the RISC iX version executes a command which is functionally equivalent to:

ld <flags> <startup> <objects/libraries> <standard library>

<flags>::	loader flags from the cc command line
<startup>::	`/lib/crt0.o` (normally)
	`/lib/mcrt0.o` (if `-p` or `-pg`)
	`/lib/gcrt0.o` (if `-g`)
<objects>::	From the command line, as and cc (ccom)
<libraries>::	From the command line
<standard library>::	`-lc` (normally and `-g`)
	`-lc_p` (if `-p` or `-pg`)

Notice that selecting a profiling option (`-p` or `-pg`) causes the startup sequence and the standard library to change. The profiled libc library (`/usr/lib/libc_p.a`) is not shared – so profiling code disables the use of a shared library. Notice also that the standard library is always at the end and that the order of the *<objects/libraries>* entries are the same as the order in which the corresponding arguments occurred on the command line – this can have a fundamental effect on the link step as `ld` is order sensitive (see below).

None of the above is affected by the use (or not) of shared libraries, however it will be seen that it is impossible to disable the use of a shared library in non-profiled code. This is intentional, it discourages (prevents) people who do not know what they are doing from circumventing the shared library mechanism. It can also mean that programs which link on other systems fail to link on a RISC iX system because of an attempt to redefine part of the shared library in the user program (`ld` will output a *multiple definition* message for the symbol in question). Normally this is a potential error, on occasions it may be an intentional attempt to change the behaviour of the standard library. If this is the case it will be necessary to link with the non-shared version of the library using `ld(1)` – see below.

ld(1)

The UNIX link editor `ld(1)` is a simple two pass linker for object modules in `a.out(5)` format and archives of such modules. In the RISC iX system it is extended to deal with the modified RISC iX version of the `a.out` header, including dealing with shared libraries. Many of the problems which programmers encounter while using `ld` are simply a result of an inadequate user model of how the linker behaves, yet `ld` actually uses an extremely simple (some would say simplistic) mechanism.

How ld works

`ld` scans its arguments twice. The first time it accumulates information from the arguments, on the second pass it writes the output file (regardless of errors in the first pass).

The information from the first pass is obtained by parsing the flags and reading the symbol tables of the object modules and archives. This is done in the order in which the arguments occur on the command line. When an archive is encountered each member of the archive is examined, if it contains a definition of a symbol which is currently undefined the archive member is read in. This may result in more undefined symbols which can be satisfied by other archive members, members are accumulated until no more symbols can be resolved. For this to work correctly the archive must be provided with a 'table of contents' using `ranlib(1)`.

If an archive has not been subjected to `ranlib`, or the table of contents is out of date, `ld` will output a warning message and make a half-hearted attempt to accumulate the relevant information – it makes a single pass over the archive accumulating entries which define unresolved symbols. The latter mechanism will successfully resolve all the required symbols if the archive has been ordered using `lorder(1)` and `tsort(1)`, but this requires a library with no cyclic dependencies and gives no advantage over using `ranlib` except that the load of the archive is slightly faster.

Because the pass over the arguments is done in the order in which they occur on the command line if an archive which defines a particular symbol occurs to the left of the archive or object module which requires that symbol the relevant archive member will not necessarily be included (it may be included because of another symbol). Thus in this case the link will probably fail with an undefined symbol. The programmer must take care to ensure that if module or archive 'A' depends on module or archive 'B' then A occurs after B on the command line. If there is a cyclic dependency involving an archive then it may be impossible to link the resultant program without extracting the relevant members from the archive and linking with them explicitly. Cyclic dependencies can, however, exist within `ranlibed` archives.

The C compiler gets the things which it tells `ld` about in the correct place. The startup `?rt0.o` module appears first and is explicitly included in the link (this is the module with the entry point and the reference to the `_main` symbol) and the standard library appears at the end. The standard library has no external dependencies. If you use `ld` you must do this yourself; try putting `-lc` at the start rather than the end to see why.

When flags are encountered during the first pass they are immediately taken into account. All but one of the flags can occur anywhere in the list of arguments. The exception is –A, the flag which defines the production of an incremental load image. Notice that older versions of `ld` are more discerning about argument position – several arguments which should be able to occur anywhere have been restricted by the command line parser to occur before any input modules, thus it is generally safer to put all the position independent arguments at the start. Position dependent flags take effect where they are encountered, in particular –u only enters an undefined symbol into the symbol table when it is encountered and –D pads the current end of the data segment; so it may appear several times.

At the end of the first pass the linker knows if any symbols remain undefined. If it is trying to produce an executable program or a shared library (as opposed to a relocatable object module which may be used as subsequent linker input) it flags an error and lists the undefined symbols. Common symbols are a special hack; any undefined symbol which has an associated value is assumed to be a common symbol

(the name refers to FORTRAN *named* COMMON). The value is assumed to be the size of the area required for the symbol and the symbol is assumed to be a data segment symbol. The linker keeps track of the maximum size requested for such a symbol and uses that. Notice that, although common symbols behave as undefined symbols during the first pass (ie they fetch in archive members which offer a definition but do not pull in members which simply offer additional common definitions of the same symbol) they are not faulted if they remain undefined at the end of the first pass – the linker simply allocates the required space.

The linker performs some very crude type checking on common symbols; it will not allow a text symbol to define the value of a common symbol. It will use the value of a data symbol or an absolute (non-relocatable) symbol. To deal with shared libraries the checking against absolute symbols is slightly modified (see below). This checking provides protection against the common error of declaring a data area symbol which clashes with a library function (in particular a `libc` function). For example:

```
int fopen;
```

will cause a multiple definition from `libc` when linked with it. Unfortunately `ld` does not spot the (equally wrong) reverse of this situation – where the function `fopen` is defined before the common symbol. `ld` is quite happy to merge a common symbol with anything. This has a particularly disastrous effect in incremental linking, fortunately modules used in this way are normally machine generated and therefore the machine can perform the required checks. If a program generating incrementally loaded object modules allows common symbols defined with user supplied names it must know the full set of symbols in the incremental base and disallow clashing common symbols).

The real work is done during the second pass. The arguments, and the corresponding modules, are read again and the text and data segments of the relevant modules are written into the output after performing suitable relocation according to the information in the relocation tables. If there were undefined symbols at the end of the first pass relocation is disabled, if it is not disabled then the relocation information (which is expressed in terms of symbol table entries) must be completely resolved.

`ld`'s favourite message during the second pass is:

<*symbol*>: <*module*>: Multiply defined

where '<*symbol*>' is the name of some symbol in input module '<*module*>'. It outputs this message whenever the definition of a symbol during the second pass generates different information from that obtained from the first pass. Normally this means that the symbol really is multiply defined, but it can also arise if `ld` manages to generate different module sizes during the second pass, either because of a bug in `ld` or because a file changed between the first and second pass.

Several symbols are built in to `ld` and these are exceptions to the above rules. `ld` will not allow most of these to be defined – they are produced when linking a program by the linker and hold details of the extents of the various segments in the program. The symbols are:

_etext	The end of the text segment of the program
_edata	The end of the program static data
_end	The end of the program data (static data plus bss)

Each of these symbols points to the byte beyond the segment in question. These symbols are not defined when a shared library is linked. In addition to these two more symbols define details of the shared libraries used by the program:

_estext	End (top) of the shared library text
_esdata	End (bottom) of the shared library data.

_estext is the byte beyond the end of the library text, _esdata however is the first byte of the shared library data (the end is defined by utop – the top of user memory). When building a new shared library these symbols may be defined by a library which the shared library shares; in this case they define the base of the text and data of the new shared library and are redefined in the output of the linker to hold the new ends. These are the only symbols which are ever redefined by `ld`.

Linking with shared libraries

When `ld` encounters a shared library, or a reference to a shared library (see `mkshare(1)` below) during its first pass it accumulates the symbols from the shared library as normal. However, these symbols are not relocatable – they are absolute locations in the shared library code or data. Thus the linker converts these symbols into absolute symbols (symbols may either be absolute, text or data). During the second pass the linker simply ignores the shared library, apart from its checks for multiple definition done by rereading the library symbols and checking that they are as expected.

If the shared library (reference) is in an archive it will be ignored unless it provides definitions which resolve undefined symbols. Shared library symbols may satisfy common symbols if the shared library symbol apparently refers to data. This is determined by examining the `a.out` header of the library and seeing if the value of an absolute symbol is within the data segment of the library (remember that the data is stored at the top of the virtual address space). This is only necessary for symbols which come from shared libraries which the library itself shares The convention to absolute symbols occurs when the library is read in, not when it is produced, so only library symbols read in during the build of the library now being loaded will already have been converted to absolute. As a further complication an incremental load (–A) base image may contain a reference to a shared library. `ld` currently insists that this is loaded first (although this is not technically necessary) so there will be no outstanding common definitions. (This is the only reason for maintaining this restriction on the command line position of the –A flag).

`ld` expects to encounter no more than one shared library or incremental base. If it encounters a second it flags an error. Thus output designed for incremental loading must never require a shared library (except in as much as the incremental base itself may). Output which requires a shared library must require only one library. `ld` places the name of this library in the `a.out` header of the result. When the kernel loads the resultant image it also obtains the shared library using the path name in the header. `ld` will produce programs with non-absolute shared library path names, but the kernel will not execute them successfully unless the path name allows the relevant library to be found. Normally this only arises if a shared library is used directly as an argument to `ld` – `ld` issues a warning message if this occurs.

As well as the name of the shared library `ld` also includes a time-stamp. The RISC iX version of `ld` always puts a time-stamp in the header of output modules unless they use the non-extended version of the `a.out` header. This time-stamp is copied from the shared library header into the header of a program which uses it. Unless the time-stamp in the shared library which the kernel finds matches that in

the program the kernel will not execute the program. The time-stamp and shared library name for a program may be examined using `phead(1)`.

Building shared libraries

A shared library is built in the same way as a normal program, except that the –Z flag is supplied to `ld`. The output must therefore be a closed system – for the link to be successful there can be no outstanding references in the linked code. Also the shared library may make a reference to a single other shared library – when a program referencing a shared library is executed the kernel loads not just the shared library but also the library which the library references, and so on.

A shared library is always demand paged (indeed, non-demand paged programs are not supported under RISC iX). As a result both the library and a referencing program must be linked so that their text segments are on suitable filing system boundaries. The data of a shared library is loaded by the kernel, it cannot be conveniently demand paged as it is in the same segment as the stack. Thus the data of the shared library immediately follows the text (with no padding to page or block boundaries) and any shared library text which overflows the last complete (memory) page is copied into the base of the program text.

The latter point has an important implication for the constructor of a shared library – the last partial page of a shared library is not actually shared. The page size on the R140 machine is large (32Kb) thus it is often important to construct the library so that it is just over a multiple of 32Kb in size. This is particularly important for `libc`, which consists of a large number of small functions most of which are not used in any given program. Many programs which use `libc` are substantially smaller than 32Kb, it is quite conceivable that a bad choice of modules for inclusion into the shared part of the `libc` would actually increase both the physical memory occupancy and the disc occupancy of the majority of `libc` applications. Equally there is never any point in sharing library code unless the resultant shared library is at least 32Kb in size (including the overflow from any library which the shared library uses.)

Some of these points are discussed in more detail in the next chapter, which attempts to define how shared libraries should be used.

mkshare(1)

A shared library cannot be conviently included in an `ar(1)` archive, although *ar* will do this the result is inefficient, (the disc space occupancy is enormous – far more than the shared library itself). It is impossible to link with the result because `ld` cannot generate a reasonable name for the library (to insert in the `a.out` header).

Despite this it is normal to insert references to shared libraries into archives, so that they look exactly like the original non-shared library. (There will normally be archive members in the archive which have not been linked into the library, as explained above).

The `mkshare(1)` program allows a shared library to be processed to generate an archivable object module which contains all the information which `ld` requires from the shared library, including the name for the shared library which should be inserted into a linked program. Conventions on shared library naming allow multiple shared libraries for the same base library to exist in a single machine. Basically the library time-stamp is used to generate a suitable unique name. This allows code using enhanced or bug-fixed libraries to exist with code which still uses the old library. `mkshare` supports this by allowing an arbitrary choice for the relevant name, and providing support to generate the name from the shared library itself.

The result of using `mkshare` is an object module with magic number 0407 (OMAGIC) which may be passed to `ld(1)` or may be archived using `ar` and `ranlib`. The object module contains all the symbols from the shared library plus that part of the shared library text which will be required to link a reference to the shared library into a program (or other shared library) and the shared library data.

The argument to `mkshare` is the single shared library to which the reference will be made. This library is not altered.

`mkshare` implements a naming scheme for shared libraries which allows the link time of the library to be built into the name – thus different instances of the same library can exist at the same time. The –i flag causes the name of the library to be formed from the specified base name (from –n, or the input file name) plus a time stamp.

By convention shared libraries are stored in `/lib` – they should normally be stored in the root partition of the machine, as otherwise programs using them will not be executable until the partition(s) containing the relevant libraries are mounted. Normally the library name will be *<rune>:<time-stamp>* for a library lib*<rune>*.a, thus the various versions of the `libc` shared library are stored in files with names of the form `/lib/c:`*<time-stamp>*.

Options

mkshare supports the following options:

`-n` This option is followed by the full path name of the shared library – the reference to the library will contain this path name (rather than the library name specified on the command line).

`-i` Add a unique id based on the shared library's time-stamp to the end of the name used for the shared library (specified by `-n` or from the input file name). The id is simply the time-stamp in yymmddhhmm.ss format.

`-p` Instead of making the reference just print the shared library name – this is intended for use with `-i` to find the full name for installation of a shared library.

`-o` This specifies the name of the reference. If not given this defaults to the library name with the trailing '`.?`' replaced by '`.o`' or (if the library name does not end in '`.?`') with the suffix '`.o`' appended.

phead(1)

`phead` prints the information in the exec header of the given object modules, executable images or shared libraries. It simply reads a list of object modules specified on the command line and, for each one, decodes and prints the information in the `a.out(5)` header.

a.out(5)

The standard UNIX `a.out` header is extended in RISC iX to deal with the requirements of shared libraries and squeezed images. The RISC iX version of the header starts with the standard header, but then contains further fields. The new fields are documented in the man page for *a.out* and in the include file obtained by:

```
#include <a.out.h>
```

The following details are extracted from the man page. Notice the change in interpretation of the a_entry field for a shared library. Layout information as obtained from the include file for the ARM is:

```
/*
* Header prepended to each a.out file.
*/
struct exec {
long            a_magic; /* magic number */
unsigned long   a_text;  /* size of text segment */
unsigned long   a_data;  /* size of initialised data */
unsigned long   a_bss;   /* size of uninitialised data */
unsigned long   a_syms;  /* size of symbol table */
unsigned long   a_entry; /* entry point */
unsigned long   a_trsize;/* size of text relocation */
unsigned long   a_drsize;/* size of data relocation */
};


/*
* Header prepended to each a.out except for those of type
* OMAGIC - note that this starts with struct exec.
*/
#define         SHLIBLEN 60

struct exec_header {
struct exec     a_exec;  /* The (old) small header */
struct version  a_version;/* Version number time and text */
unsigned long   a_sq4items;/* number of squeezed type 4 items */
unsigned long   a_sq3items;/* number of squeezed type 3 items */
unsigned long   a_sq4size;/* size of squeezed type 4 items */
unsigned long   a_sq3size;/* size of squeezed type 3 items */
unsigned long   a_sq4last;/* last entry in type 4 table (check only) */
unsigned long   a_sq3last;/* last entry in type 3 table (check only) */
time_t          a_timestamp;/* link time of this object */
time_t          a_shlibtime;/* time-stamp of shared library */
char            a_shlibname[SHLIBLEN];/* Path name of shared library */
};


/*
* Basic magic numbers
*/
#define         OMAGIC  0407       /* old impure format */
#define         NMAGIC  0410       /* read-only text */
#define         ZMAGIC  0413       /* demand load format */


/*
* Flags which may be ored with the magic number.
* All combinations are valid.
*/
#define MF_IMPURE00200    /* impure text */
#define MF_SQUEEZED01000  /* text and data squeezed */
#define MF_USES_SL02000   /* this object uses a shared library */
#define MF_IS_SL 04000    /* this object is a shared library */


/*
* Names for common combinations
*/
#define IMAGIC    (MF_IMPURE|ZMAGIC)  /* demand load format (impure text) */
#define SPOMAGIC  (MF_USES_SL|OMAGIC) /* OMAGIC with a large header - may */
                  /* contain a reference to a shared*/
                  /* library required by the object.*/
#define SLOMAGIC  (MF_IS_SL|OMAGIC)   /* a reference to a shared library. */
                  /* The text portion of the object*/
                  /* contains "overflow text" from*/
                  /* the shared library to be linked*/
                  /* in with an object.*/
#define QMAGIC    (MF_SQUEEZED|ZMAGIC)/* squeezed demand paged*/
#define SPZMAGIC  (MF_USES_SL|ZMAGIC) /* program which uses shared lib*/
#define SPQMAGIC  (MF_USES_SL|QMAGIC) /* squeezed ditto*/
#define SLZMAGIC  (MF_IS_SL|ZMAGIC)   /* shared library part of program*/
#define SLPZMAGIC (MF_USES_SL|SLZMAGIC)/* shared lib which uses another*/
```

Squeezed shared libraries are not sorted:

```
/*
* Macros which take exec structures as arguments and tell whether
* the file has a reasonable magic number or give offsets to
* text | symbols | strings.
.*/
#define N_BADMAG(x) \
    ( ( ((x).a_magic & ~007200) != ZMAGIC ) && \
      ( ((x).a_magic & ~006000) != OMAGIC ) && \
      (  (x).a_magic != NMAGIC ) \
    )


#define          IS_SQUEEZED(magic)   (((magic) & MF_SQUEEZED) != 0)
#define          IS_SHARED_LIB(magic) (((magic) & MF_IS_SL)    != 0)


#define N_TXTOFF(x) \
    (     (x).a_magic == OMAGIC ? sizeof (struct exec) : \
          (((x).a_magic & ~007200) == ZMAGIC ? PAGESIZE : sizeof (struct exec_header)) \
    )
#define N_SYMOFF(x) \
        (N_TXTOFF(x) + (x).a_text+(x).a_data + (x).a_trsize+(x).a_drsize)
#define N_STROFF(x) \
        (N_SYMOFF(x) + (x).a_syms)
```

The file has five sections: a header, the program text and data, relocation information, a symbol table and a string table (in that order). The last three may be omitted if the program was loaded with the '`-s`' option of `ld` or if the symbols and relocation have been removed by `strip`(1).

In the header the sizes of each section are given in bytes. The size of the header is not included in any of the other sizes.

When an `a.out` file is executed, three logical segments are set up: the text segment, the data segment (with uninitialised data, which starts off as all 0, following initialised), and a stack. Only 'demand load' (ZMAGIC) formats may be loaded – the other formats are historical relics (NMAGIC) or are input files for `ld` (OMAGIC).

The text segment begins at offset 32768 (the start of the first page) in the virtual memory of the process. For convenience during loading the text segment begins at offset 32768 in a ZMAGIC file (see the definition of N_TXTOFF above.) For a program which does not use a shared library (the MF_USES_SL flag is not set in the magic number) this is loaded at virtual address 32768. Otherwise it is loaded at the first page after the end of any shared library text (see the description of shared libraries below.) Unless the 'impure' (MF_IMPURE) flag is set the text segment will not be writable and will be shared between different instances of the same program. If the file is impure the text segment size will be 0 – `ld` merges the text and data segments.

The data segment begins immediately after the text segment. The linker rounds the size of the text segment up to a 32768 byte boundary, thus the data segment starts on a (memory) page boundary in the file (this also happens to be a file system page boundary). The data segment is loaded immediately after the text segment and is, of source, writable. The uninitialised (`bss`) data occurs immediately after this and is initialised to zero. See the description of `ld` for details of the symbols which may be used in a program to determine the virtual address of these segments of the address space.

The heap occurs immediately after the end of the uninitialised data – this may not be on a memory page boundary. The stack segment is at the top of the virtual address space, and also contains the shared library data (see below). This segment ends at USRSTACK (from *<machine/vmparam.h>*). The stack is automatically extended as required. The data segment is only extended as requested by `brk`(2).

If the image is squeezed (the MF_SQUEEZED flag is set in the magic number) the text and data segments in the `a.out` file will be squeezed. When the kernel loads the relevant memory pages the contents of the relevant section of the file will be unsqueezed. In this case the first 32768 bytes of the file (corresponding to the first memory page) contain the tables for unsqueezing the remainder of the file. The extended exec header contains the information the kernel requires to locate these tables.

If the magic number in the header is identically OMAGIC (0407) the old format exec header is used – this is for compatibility with programs producing input for the linker. For all other formats the extended header is used. This contains information about the version of the program or binary, the squeeze tables (as above), the link time and any shared library requirement.

The version information is left blank by `ld`. It is filled in separately for all distributed binaries and may be read using `version`(1).

The squeeze information is also left blank – it is filled in by the program which squeezes the linked image.

The *a_timestamp* field is filled in by `ld`, which sets it to the link time of the output. The shared library information is filled in at the same time – for a program which does not use a shared library it is blank (all bytes 0) for programs which do it contains the time-stamp from the shared library and the absolute path name of the shared library.

Shared libraries are indicated by the MF_IS_SL flag in the magic number and are treated very differently by the kernel. A shared library is produced using the `-z` flag to `ld`(1). It may not be executed directly but may be used in a further link step to produce a program which is capable of sharing the library code with other programs linked with the shared library – see the description of `ld`(1) and `mkshare`(1).

When the kernel encounters a program which uses a shared library it looks for the library – the program cannot be executed unless the library is in the correct place. Use `file`(1) or `phead`(1) to find out details of the shared library used by a program.

The kernel loads the shared library text first (recursively loading a shared library used by the library if necessary.) The data for the shared library is placed at the virtual address given by the *a_entry* field in the shared library exec header. `Ld` sets this so that the data sits immediately below USRSTACK, or below the data of a shared library which the library shares. Shared libraries never have any uninitialised data. The program text and data is loaded after the shared library text. The kernel then commences execution of the process at the entry point given by the *a_entry* field in the exec header of the program.

After the header in the file follow the text, data, text relocation data relocation, symbol table and string table in that order. The N_TXTOFF macro returns this absolute file position of the text segment when given the name of an exec structure as argument. The data segment is contiguous with the text and immediately followed by the text relocation and then the data relocation information. The symbol table

follows all this; its position is computed by the N_SYMOFF macro. Finally, the string table immediately follows the symbol table at a position which can be gotten easily using N_STROFF. The first four bytes of the string table are not used for string storage, but rather contain the size of the string table; this size INCLUDES the four bytes, the minimum string table size is thus four.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file as follows:

```
/*
* Format of a symbol table entry.
*/
struct nlist {
                union {
                        char        *n_name;    /* for use when in-core */
                        long        n_strx;     /* index into file string table */
                } n_un;
                unsigned charn_type;/* type flag, i.e. N_TEXT etc; see below */
                char    n_other;
                short   n_desc;     /* see <stab.h> */
                unsigned longn_value;/* value of this symbol (or offset) */
};
#define         n_hash  n_desc      /* used internally by ld */

/*
* Simple values for n_type.
*/
#define         N_UNDF  0x0         /* undefined */
#define         N_ABS   0x2         /* absolute */
#define         N_TEXT  0x4         /* text */
#define         N_DATA  0x6         /* data */
#define         N_BSS   0x8         /* bss */
#define         N_COMM  0x12        /* common (internal to ld) */
#define         N_FN    0x1f        /* file name symbol */


#define         N_EXT   01          /* external bit, ored in */
#define         N_TYPE  0x1e        /* mask for all the type bits */


/*
* Other permanent symbol table entries have some of the N_STAB bits set.
* These are given in <stab.h>
*/
#define         N_STAB  0xe0        /* if any of these bits set, don't discard */


/*
* Format for namelist values.
*/
#define         N_FORMAT "%08x"
```

In the `a.out` file a symbol's n_un.n_strx field gives an index into the string table. A n_strx value of 0 indicates that no name is associated with a particular symbol table entry. The field n_un.n_name can be used to refer to the symbol name only if the program sets this up using n_strx and appropriate data from the string table.

If a symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader `ld` as the name of a common region whose size is indicated by the value of the symbol.

The value of a byte in the text or data which is not a portion of a reference to an undefined external symbol is exactly that value which will appear in memory when the file is executed. If a byte in the text or data involves a reference to an undefined external symbol, as indicated by the relocation information, then the value stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the bytes in the file.

If relocation information is present, it amounts to eight bytes per relocatable datum as in the following structure:

```
/*
* Format of a relocation datum.
*/
struct relocation_info {
int             r_address;/* address which is relocated */
unsigned int    r_symbolnum:24,/* local symbol ordinal */
                r_pcrel:1, /* was relocated pc relative already */
                r_length:2,/* 0=byte, 1=word, 2=long */
                r_extern:1,/* does not include value of sym referenced */
                r_neg:1, /* negative relocation */
                :3;     /* nothing, yet */
};
```

There is no relocation information if a_trsize+a_drsize==0. If r_extern is 0, then r_symbolnum is actually a n_type for the relocation (ie N_TEXT meaning relative to segment text origin).

See also

adb(1), as(1), ld(1), mkshare(1), nm(1), dbx(1), stab(5), strip(1), squeeze(1), unsqueeze(1).

Bugs

The a_entry symbol is overloaded.

Although all magic number/flag combinations are valid most of them are not understood by the kernel – in particular magic numbers based on OMAGIC or NMAGIC will not execute correctly. All valid combinations are listed in the include file.

6.3 Guidelines

Normally the use of a shared library should be transparent to the user. When it is not the user may question the use of a shared library. Normally this arises when an application program attempts to redefine symbols in the shared library, and normally this is not a good idea. There are three basic circumstances:

Using shared libraries

1 A genuine error – the application programmer was not aware that such a symbol existed in the library. This should be corrected. It can be avoided by using the standard techniques for ensuring symbol uniqueness; typically the program 'prog' only defines symbols of the form 'prog_<name>'. This is often extended in large packages to use a variety of prefixes.

2 An attempt to ensure portability. For example by implementing a function which exists in some operating systems but not others. In this case the implementation should be removed conditionally in those environments in which it is already provided (for example by use of C preprocessor directives or by excluding the relevant modules from the link stage).

3 Intentionally extending or altering the functionality of parts of the library. Normally this should be done by avoiding the relevant part of the library and using a different interface with a different name. Sometimes the application programmer may wish to change a function which is actually used in the library – so that the library uses the modified implementation. Unless the interface the library requires is well defined this is very error prone.

It is easier to avoid these circumstances than to deal with them when they arise. Working in an ANSI C environment (cc -ansi) helps to avoid (2). ANSI defines a set of interfaces which are provided on all ANSI implementations. In general if common functions (eg memcpy) are not available it is better to implement them separately for each system, linking the application with a system specific module,

than to attempt to implement them for all systems. It is unlikely that you will be able to provide the correct interface on all systems, for example the RISC iX `bcopy` function returns a result – some parts of the `libc` library rely on this. The internal interfaces of `libc` in particular are completely undefined.

Building shared libraries

This section gives guidelines on building a shared library.

Some generalisations can be made about code which is unsuitable for sharing. Archive members which define a lot of static data are generally unsuitable for sharing unless a program which uses the shared library will also almost always use the member in question. If this is not the case programs will acquire large amounts of static data which is never used – this will have a detrimental effect on their memory occupancy. Bear in mind that the R140 system has a restricted number of pages and amount of swap space. There are around 70 pages (a very rough figure) available for running processes, and the swap space is only 8Mb (by default). The UNIX kernel allocates a page of swap space for each page of data regardless of whether it ever needs to swap that data. If a program holds a page of data which is never used, swap will be allocated for it, even though physical memory never will (it is demand loaded). If a program has an unused page of shared data the situation is the same, except that a memory page is allocated initially (because the kernel loads shared data at load time – it is not demand loaded).

Library archive members with undefined symbols which cannot be resolved when the shared library is linked can never be shared. Sometimes it is worthwhile to work round this. For example important parts of `libc` reference the _end symbol to find the end of the program data – obviously this is not known until the program itself is linked. To deal with this a pointer variable _pend is declared and the value from this used instead of _end. The startup module `crt0.o` must fill in this and other program specific values.

Library members which are used in relatively few of the programs which link with the library are normally not worth sharing. Exceptions to this rule are, however, quite common. If the library members require no static data, or an insignificant amount of static data, the cost of sharing them is normally small. If the library text size would otherwise be just under a page (32Kb) the cost of not sharing them would actually be large! Sharing is beneficial in terms of disc occupancy even if only two programs share the library members. For there to be a benefit in terms of memory occupancy two programs sharing the same code must execute concurrently, but if one of the programs which uses those library elements is always executing when the library is in use by any program there is no cost. In practice the decision about whether to include a particular library member is best made by including the most frequently used members, and those other members which are required to resolve undefined symbols, then including less frequently used members until the text size is just over a page boundary.

If a library uses a large amount of static data it can still be worth sharing it. A good example is the `libXt` library, which implements a crude form of object oriented programming by using statically allocated object classes. The style of programming adopted means that inclusion of one part of `Xt` normally includes many other parts – all the ancestors of the included class. `libXt` defines the generic classes, other libraries built on top of it define derived classes of objects. It is worthwhile sharing a substantial part of `libXt` as a result – the static data size of the shared library is large but the programs sharing it would require this static data even if it was not shared.

Shared libraries can be particularly advantageous on a system which is prone to swapping if the programs in question spend a lot of time in the shared code. This is a relatively rare situation, most applications spend a relatively small amount of time in library code. It is possible, however, to conceive of an application suite which can be built in such a way as to cause significant processing in sharable code. In this case it might be worthwhile to rewrite the code in question so that it can be built into a shared library.

An example

Suppose the archive `libX11.a` contains the current (non-shared) version of the X11 library and that this is to be made shared. The first step is to rename `libX11.a` to `libX11_n.a` – the non-shared library will be preserved so that it can be linked with -lX11_n if any application should require this. Then the list of modules which will form the shared library should be worked out – this will probably be an iterative process as the first attempt will form a library which is the wrong size, or has too much static data. Suppose the file `sharedfiles.dat` contains the list of files:

```
ar x libX11_n.a
ld -o libX11.S -Z `cat sharedfiles.dat` -lm -lc
```

Notice that the maths library `libm.a` is required (this is not shared) and that the shared C library is used as a base. Thus some parts of `libm` and possibly some otherwise unshared parts of `libc` will be incorporated into the new library. This is the stage at which it is found that `libX11.a` has references to other parts of the system – this may or may not prevent further progress. If the other required libraries are already shared then, if they are built on top of the shared *libc*, there is no problem. The libraries are placed on the `ld` command line before `-lc` and provide all the symbols which would otherwise come from the shared parts of *libc*. If the other requirements are actually provided by the application program it will be necessary to prune the modules from `sharedfiles.dat` which reference these parts. If this is not possible the libraries is not suitable for sharing.

A good example of a library which cannot be shared is one which uses *yacc*(1) generated source but requires that the user (ie application programmer) provide the `yylex` function. The *yacc* parser will have an unresolved reference to `yylex` which is only resolved when the application is linked. Such a library can be rewritten by defining a `yylex` which indirects via a static function variable – then the application programmer would set the variable to a suitable function before calling the parser, however this would change the library interface.

When the `libX11.S` shared library has been produced it can be examined using `phead`(1) (see above):

```
libX11.S:
                ZMAGIC shared library [requires shared library]
                text size: 69176(10e38)
                data size: 11416(2c98)
                bss size:  0(0)
                sym size:  21564(543c)
                data base: 16733032(ff5368)
                text relc: 0(0)
                data relc: 0(0)
                timestamp: Thu Apr  6 16:46:43 1989
                needs library: '/lib/c:8903041822.48' (time Tue Apr  4 19:22:48 1989)
```

This may be satisfactory – the text overflow is 3640 bytes over a page, and the 11Kb of data is probably acceptable (this figure includes the data used by the shared *libc* library – in this case 4Kb). If not some fine tuning can be performed, using `phead`(1) or `size`(1) on the input object modules. (`size`(1) gives the size of the

text, data and bss segments of any object module, `phead` gives additional information about the time–stamp and shared library, as well as decoding the shared library entry point data).

Once a suitable shared library has been obtained `mkshare` can be used to generate a suitable reference to it. By convention the reference module is called `libX11.o` – ie with the `.a` of the archive replaced by `.o`. Similarly the shared library itself has the extension `.S` while it is being built, however it is installed with the name generated by `mkshare`. Thus:

```
mkshare -o libX11.o -ni /lib/X11: libX11.S
```

Then the installation step can be completed by using *mkshare* to print the name of the file. Notice that this allows the use of these commands in a `2z2zMakefile` for processing by `make`(1) – otherwise the name of the installed library has to be obtained by examining the output of `phead`(1).

```
install -c -m 555 -o root.wheel libX11.S `mkshare -nip /lib/X11: libX11.S`
```

Notice that the shared library must be executable – the kernel requires this. The use of no write access and ownership by the superuser is also very advisable. See the comments at the beginning of this document about destroying your RISC iX system.

You will not be able to test the shared library until you have installed it, unless you link programs with it directly. To do this work out the absolute path name of the shared library and link (for example):

```
ld -o prog /lib/crt0.o prog.o /zigguratusr/x11r2/lib/X/libX11.S -lX11_n -lc
```

Chapter 7 – Squeezed image files

Squeezed image files

7.1 Introduction

One problem with Reduced Instruction Set processors is that they tend to have a *bulky* instruction coding. In order to make better use of the disc space available on low end RISC iX machines a compression technique has been employed. This technique has the advantage of providing lower disc occupancy and faster loading from disc or net (but note that programs which understand the structure of a.out files may only work on uncompressed images).

RISC iX provides a new class of executable binaries called *squeezed demand paged* which allow the text and initialised data areas to be stored in a compressed form such that the kernel can still demand page from the text area of the program file.

Two utilities `squeeze(1)` and `unsqueeze(1)` are provided to convert between the compressed and ordinary forms: it is a user decision to compress a file (usually after the program in question has been fully debugged and is to be installed permanently in the filing system). Most of the RISC iX toolkit is supplied by Acorn in compressed form.

7.2 Compression algorithms

There are two alternatives for file compression:

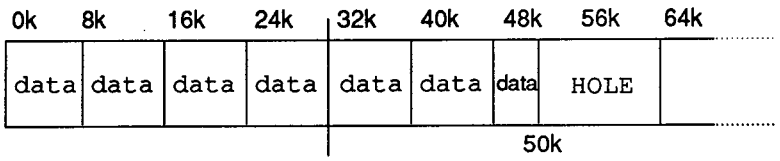
- A general purpose method, for example that provided by the BSD compress utility which uses Adaptive Lempel-Ziv encoding. This method is unsuitable because although it achieves very good results it uses variable size compression tables and arbitary (9–16) bit encodings which can occupy considerable space and take substantial CPU time to decompress.
- A compression technique optimised for fast decompression and the ARM instruction set. The algorithm developed uses byte data, and a maximum 14Kb data table.

Page (32Kb) chunks verses a total squeeze

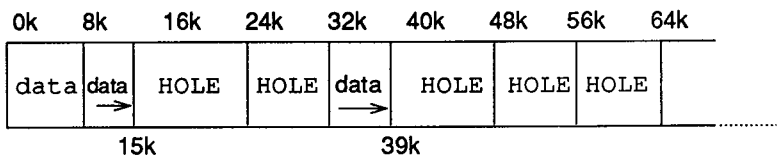
The RISC iX filesystem supports sparse files, that is a file containing a large (greater than one 8Kbyte disc block) *hole* of unwritten data (generated by seeking) will not occupy any space on the disc. The filing system ensures that the user reads the *hole* as all zeros. The technique employed utilises this feature by compressing the file to a series of chunks each of which when uncompressed will occupy exactly one MEMC page of 32Kbytes. If, for example, a file has 50Kbytes in its text area, this is two pages. Suppose each can be compressed: The first 32K to 15Kbytes, the remaining 20K to 7Kbytes. The compressed forms are placed in the file on 32Kbyte boundaries to make extraction simple and changes to the kernel minimal (it currently demand pages 32Kbyte pages from 32Kbyte boundaries). The disc space occupancy is only 2*8 + 1*8 = 24 Kbytes since the rest of the file is comprised of *holes*.

File layout

Before compression:



After compression:



This *chunky* technique has the advantage that it integrates into the kernel paging system extremely easily. It provides approximatly a 37% reduction in disc occupancy (measured on files in `/bin`).

This method does not achieve maximal compression, for if a compressed chunk opfccupies 16.5K it only saves one disc block (since unused areas must be .8Kbyte aligned). An alternative technique would be to compresses the entire file and generate a table of pointers to the 32Kbyte boundaries. This makes extraction more difficult (since the kernel must access the file on non block boundaries, and this conflicts with existing data structures) and so it has not been pursued. Studies have shown that it is likely to provide another 15% reduction in disc occupancy. Other avenues include reducing the disc block size to 4K or 2K.

7.4 The encoding scheme

The scheme adopted uses an extended a.out format, which stores tables which can be used to expand the compressed text section of an executable file.

1 For space-efficient decoding (since the compressed form will be smaller than the uncompressed we can do some decompression in place), compressed bytes are read in descending address order (ie 'next byte' means *byte_pointer–).

Each word (four bytes) is encoded in one of four ways, the encoding being specified by a 4 bit value (nibble) thus.

nibble = 0 The word is zero

nibble = 1 The word is given in full as the *next* four bytes, low byte first.

nibble = 2..8 *long*: The top three bytes of the word are obtained by taking ((nibble – 2) << 8) | next byte as an index into a table of up to 1792 common values (*longs* table). Then the low byte is given explicitly as the next byte.

nibble = 9..15 *short*: ((nibble–9) << 8) | next byte gives index into a table of up to 1792 common word values (the *shorts* table).

The nibbles describing the encoding of a pair of words (in ascending address order, eg first word at a, second at a+4) are packed together into a single byte, as first | (second << 4), so the sequence is thus:

/* bytes in descending address order */
1 byte [first | (second << 4)]

[0, 1, 2, or 4 bytes giving additional specification of first word]
[0, 1, 2, or 4 bytes giving additional specification of second word]

The image will be padded with an arbitrary word to make it a multiple of eight bytes.

2 The *shorts* table is a table of word values, in ascending order (of unsigned comparison, ie 0..FFFFFFFF). It is stored on disc in a compressed form encoded thus:

For each entry j, j >= 0, one byte specifying the encoding thus:
(let the encoding byte be b0 and successive bytes be b1 b2 b3 ...)

```
b0 = 0        table[j] = table[j-1] + ((b4<<24) | (b3<<16) | (b2<<8) | b1)
b0 = 1..9     table[j]..table[j+(b0-1)] = (table[j-1]+1)..(table[j-1]+b0)
b0 = 10..91   table[j] = table[j-1] + (b0-10)
b0 = 92..173  table[j] = table[j-1] + (((b0-92) << 8) | b1)
b0 = 174..255 table[j] = table[j-1] + (((b0-174) << 16) | (b2 << 8) | b1)
```

[We take table[–1] =–1 as the starting point].

- The longs table is encoded almost the same as the shorts table, except that table[j] holds (encoded_value >>8), since the low bytes of all the entries are zero, and in the case b0 = 0, we only have three succeeding bytes, not four.

As a check the last entries of the long and short tables is stored in the fileheader.

Format of a compressed executable file

Below are the structures (extracted from a.out.h) defining the format of an executable image:

```
struct exec {
    long            a_magic;     /* magic number */
    unsigned long   a_text;      /* size of text segment */
    unsigned long   a_data;      /* size of initialized data */
    unsigned long   a_bss;       /* size of uninitialized data */
    unsigned long   a_syms;      /* size of symbol table */
    unsigned long   a_entry;     /* entry point */
    unsigned long   a_trsize;    /* size of text relocation */
    unsigned long   a_drsize;    /* size of data relocation */
};

struct exec_header {
    struct          exec a_exec; /* The (old) small header */
    struct  version a_version;  /* Version number time and text */
     unsigned long  a_sq4items;  /* number of 4 items (= "short") */
     unsigned long  a_sq3items;  /* number of type 3 items (="long") */
     unsigned long  a_sq4size;   /* size of squeezed type 4 items table */
     unsigned long  a_sq3size;   /* size of squeezed type 3 items table */
     unsigned long  a_sq4last;   /* last entry in type 4 table (check only) */
     unsigned long  a_sq3last;   /* last entry in type 3 table (check only) */
    time_t          a_timestamp; /* link time of this object */
    time_t          a_shlibtime; /* timestamp of shared library */
    char            a_shlibname[SHLIBLEN]; /* Path name of shared library */
};
```

Following the header there is:

1 The encoding of the entries for the *shorts* table, a table of up to 1792 (= 7*256) 4–byte words which will be encoded as the leading nibble + one extra byte (the *short* encoding).

2 Then the encoding of the entries for the *longs* table, a table of up to 1792 (= 7*256) 3–byte values.

The compressed text/data starts at offset 32K in the file. For each 32K page (or part thereof) there is:

A header:

```
short rpoffset;      /*
                      * offset (from start of this block)
                      * of last compressed data byte
                      * =0, empty page
                      */
short wpoffset;      /*
                      * offset (from start of the block)
                      * of last word of de-compressed data
                      */
unsigned checksum; /*  additive checksum of all words (uncompressed) */
```

The data to be decompressed.

Example decompression routines

The following C fragment should make the proceeding definitions clear.

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/signal.h>
#include <sys/user.h>

#define UNSQUEEZE(nib)                                                  \
{    unsigned nibble = (nib), nx;                                       \
     switch (nibble) {                                                  \
          case 0:                                                       \
                    *--wp = 0;                                          \
                    break;                                              \
          case 1:                                                       \
                    nx = *--rp;                                         \
                    nx |= (*--rp) <<  8;                                \
                    nx |= (*--rp) << 16;                                \
                    nx |= (*--rp) << 24;                                \
                    *--wp = nx;                                         \
                    break;                                              \
          case 2: case 3: case 4:                                       \
          case 5: case 6: case 7: case 8:                               \
                    nx = ((nibble - 2) << 8) | *--rp;                   \
                    if (nx > sq3_items) {                               \
                              sqerr("unsqueeze: bad sq3 item", sqparam); \
                              return (1);                               \
                    }                                                   \
                    *--wp = (sq3_table[nx] << 8) | *--rp;               \
                    break;                                              \
          case 9: case 10: case 11:                                     \
          case 12: case 13: case 14: case 15:                           \
                    nx = ((nibble - 9) << 8) | *--rp;                   \
                    if (nx > sq4_items) {                               \
                              sqerr("unsqueeze: bad sq4 item", sqparam); \
                              return (1);                               \
                    }                                                   \
                    *--wp = sq4_table[nx];                              \
                    break;                                              \
     }                                                                  \
     chksum += *wp;                                                     \
}

int unsqueeze(buffer, buf2, buf2size, sqerr, sqparam,
                    sq3_table, sq3_items,
                    sq4_table, sq4_items)
     unsigned char *buffer;                /* The buffer to be unsqueezed */
     unsigned char *buf2;                  /* A working buffer for overflows */
     int buf2size;
     void *sqerr(/* char *, sqparam */);/* A function to report errors on */
     void *sqparam;                        /* An argument for sqerr() */
     int sq3_items, sq4_items;             /* Number of entries in the two tables */
     int sq3_table[], sq4_table[];         /* Desqueeze tables */
/* returns: 1 unsqueezed succeeded, 0 failed */
{
```

```
    struct info { short rpoffset;
                          short wpoffset;
                          unsigned chksum;
          } *ip = (struct info *)buffer;
    register unsigned char nibbles,
                                  *rp = buffer + ip->rpoffset;
    register unsigned *wp = (unsigned *)buffer + ip->wpoffset,
                                  chksum = 0, chksum_wanted = ip->chksum;
    if (ip->rpoffset == 0) {
                    /* This is a totally empty page ! */
                    return (1);
    }
    if (ip->wpoffset & 1) {
                    sqerr("unsqueeze: bad wp", sqparam);
                    return (0);
    }

    bcopy((caddr_t)buffer, (caddr_t)&buf2[0], buf2size);
    while ((unsigned char *)wp > rp + 4+4+4) { /* 4+4+4 = 2 words + a bit for luck */
                    nibbles = *--rp;
                    UNSQUEEZE((nibbles >> 4) & 0xf);
                    UNSQUEEZE(nibbles & 0xf);
    }

    if (wp < (unsigned *)buffer) {
                    sqerr("unsqueeze: wp overflow", sqparam);
                    return (0);
    }

    rp -= (int)buffer;
    rp += (int)&buf2[0];
    if (rp >= &buf2[buf2size]) {
                    sqerr("unsqueeze: rp underflow", sqparam);
                    return (0);
    }

    while (wp > (unsigned *)buffer) {
                    nibbles = *--rp;
                    UNSQUEEZE((nibbles >> 4) & 0xf);
                    UNSQUEEZE(nibbles & 0xf);
    }

    if (rp != (buf2 + sizeof(struct info))) {d

                    sqerr("unsqueeze: wrong amount", sqparam);
    } else if (chksum != chksum_wanted) {
                    sqerr("unsqueeze: chksum failed");
    } else
                    return (1);
    return (0);
}
int decode_squeeze_table(type, coded, nbytes, table, nitems, last, path, prerr)
    int type;                       /* 3 or 4 - just for messages */
    unsigned char *coded;/* the compressed squeeze table */
    int nbytes;                     /* size of compressed table */
    long *table;                    /* destination of decompressed table */
    unsigned nitems; /* items in table */
    int last;                       /* check value of last entry */
    char *path;                     /* path name - just for error messages */
    void *prerr(/* char *path, int type, char *errorstring */);
                                    /* error reporting function */
/* returns: 1 decompress succeeded, 0 failed */
{
    register unsigned char *cp = coded;
    long *table0 = table;
    int prev = -1, delta;
    char *err = 0;
    struct sq_unsqueeze *dummy;

    if (nitems > (sizeof(dummy->sq_table) / sizeof(long))) {
                    err = "bad table size";
                    goto bad;
    }
```

```
    while (nbytes > (cp - coded)) {
                    unsigned b = *cp++;
            if (b == 0) {
                                int b1 = *cp++;
                                int b2 = *cp++;
                                int b3 = *cp++;
                                int b4 = (type == 4) ? *cp++ : 0;
                                delta = (b4 << 24) | (b3 << 16) | (b2 << 8) | b1;
                                *table++ = (prev += delta);
                    } else if (b <= 9) {
                                while (b-- > 0) {
                                *table++ = ++prev;
                                }
                    } else if (b <= 91) {
                                delta = (b - 10);
                                *table++ = (prev += delta);
                    } else if (b <= 173) {
                                int b1 = *cp++;
                                delta = ((b - 92) << 8) | b1;
                                *table++ = (prev += delta);
                    } else /* b >= 174 */ {
                                int b1 = *cp++;
                                int b2 = *cp++;
                                delta = ((b - 174) << 16) | (b2 << 8) | b1;
                                *table++ = (prev += delta);
                    }
    }
    if (nitems != (table - table0)) {
                    err = "table wrong size";
    } else if (nitems && (last != prev)) {
                    err = "wrong last entry";
    } else
                    return (1);

bad:
    prerr(path, type, err);
    return (0);
}
```