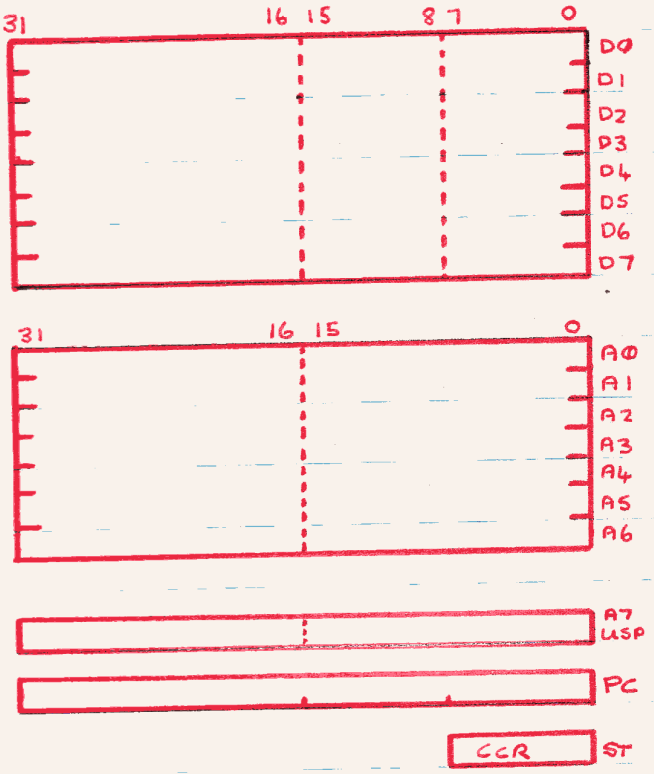
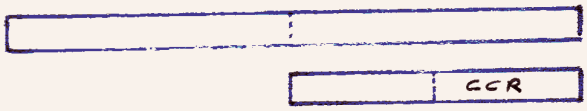


USER PROGRAMMER'S MODEL



SUPERVISOR PROGRAMMER'S MODEL (SUPPLEMENT)

M68000 Assembly Language Programming

1 Introduction

The M68000 16/32-bit processor was introduced by Motorola in 1979. It has a 16-bit data bus and 24-bit address bus, allowing direct access of 16 Mb of memory. Other processors in the family are:

M68008 - identical to program, this has a 8-bit data bus and 20-bit (1 Mb) address bus.

M68010 - M68000 with virtual machine features.

M68020 - M6800 with 32-bit data and address (4Gb) buses

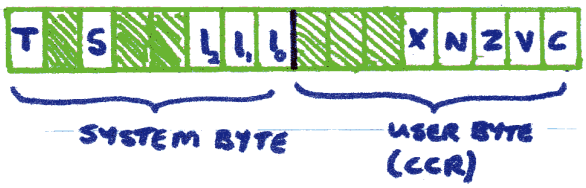
2 Programmer's Model

The M68000 executes instructions in one of two modes - user mode or supervisor mode. The supervisor mode allows some additional instructions and privileges and is intended for use by the O/S and other system software. The user programmer's model is shown opposite.

There are 16 32-bit general purpose registers (D0-D7, A0-A7), a 32-bit program counter, and an 8-bit condition code register. The data registers can be used for byte (8-bit), word (16-bit), and long word (32-bit) operands. The address registers (A0-A7) may be used as software stack pointers and base address registers, as well as word and long word operands. All of these 16 registers may be used as index registers. The supervisor model include a high-order byte of the status register and the supervisor stack pointer.

The status register contains the interrupt mask (eight levels) as well as the condition codes: overflow (V), zero (Z), negative (N), carry (C) and extend (X). Additional status bits indicate that the processor is in trace (T) mode and/or in a supervisor (S) state.

STATUS REGISTER



Five basic data types are supported, being:

- bits
- BCD digits (4 bits)
- bytes (8 bits)
- words (16 bits)
- long words (32 bits)

In addition, operations on other data types such as memory addresses, status word data, etc, are provided for in the instruction set.

There are 14 addressing modes of six basic types:

- register direct (data/addr reg direct) Dn, An
- register indirect (addr, post+, pre-, offset, indexed, d(PC)) (An), (An)+, -(An), d(An), d(An,Xn), d(PC), d(PC,Xn)
- absolute (short/long) xxx.W xxx.L
- immediate (quick) #data
- program counter relative (offset, indexed offset) pc@(d), pc@(d,Xn)
- implied. (SP, CCR, PC, all regs)

Included in the register indirect addressing modes is the capability to do post-incrementing, predecrementing, offsetting

and indexing. Program counter relative addressing can also be modified by indexing and offsetting.

3 Addressing Modes.

3.1 Register Direct Addressing

(a) Data Register Direct

AS Syntax: dn

Effective address EA = Dn

(b) Address Register Direct

AS Syntax: an

EA = An

3.2 Absolute Data Addressing

(a) Absolute Short

AS Syntax: xxx.w

EA = (next word)

(b) Absolute Long

AS Syntax: xxx.l

EA = (next two words)

3.3 Program Counter Relative Addressing

(a) Relative with Offset

AS Syntax: pc@(d16)

EA = (PC) + d16

(b) Relative with Index & Offset

AS Syntax: pc@(d8, Rx:w/l)

EA = (PC) + (Xn) + d8

3.4 Register Indirect Addressing

(a) **Register Indirect** — AS Syntax: `an@`

EA = (An)

(b) **Postincrement Register Indirect** — AS Syntax: `an@+`

EA = (An), An ← An + N*

(c) **Predecrement Register Indirect** — AS Syntax: `an@-`

An ← An − N*, EA = (An)

(d) **Register Indirect with Offset** — AS Syntax: `an@(d16)`

EA = (An) + d16

(e) **Indexed Register Indirect with Offset** — AS Syntax: `an@(d,Rx:W/L)`

EA = (An) + (Xn) + d8

3.5 Immediate Data Addressing

(a) **Immediate** — AS Syntax: `#data`

DATA = next word(s)

(b) **Quick Immediate** — AS syntax:

INHERENT DATA

3.6 Implied Addressing — AS Syntax: `sp, pc, cc, sr, usp`

IMPLIED REGISTER: EA = SR, USP, SSP, PC, VBR, SFC, DFC

* See §3.7

3.7 Symbols Used

EA ≡ effective address
An ≡ address register
Dn ≡ data register
Xn ≡ address or data register used as index register
SR ≡ status register
PC ≡ program counter
( ) ≡ contents of
d_8 ≡ 8-bit offset (displacement)
d_{16} ≡ 16-bit offset
N ≡ 1 for byte ops, 2 for word ops, 4 for long word ops, unless An is the stack pointer and the operand size is byte, in which case N=2 to keep the stack pointer on a word boundary.

4 Instructions.

The instruction set is summarised overleaf. The status columns indicate the effect on the CCR. Symbols used are:

U - undefined
* - set according to condition occurring
X,N,Z,V,C - set the same as the specified flag (eg for ABCD, the X-bit is set the same as the carry bit)
— - not affected.
• - cleared if condition does not hold, else unchanged.
0 - always cleared
1 - always set.

TYPE	VARIATION	DESCRIPTION	OPERAND TYPES 1st	OPERAND TYPES 2nd	STATUS X	N	Z	V	C
—	ABCD	Add decimal with extend	Dy -(Ay)	Dx -(Ax)	C	U	•	U	*
ADD	ADD	Add	<EA> Dn	Dn <EA>	C	*	*	*	*
	ADDA	Add address	<ea>	An	–	–	–	–	–
	ADDQ	Add quick	#<data>	<ea>	C	*	*	*	*
	ADDI	Add immediate	#<data>	<ea>	C	*	*	*	*
	ADDX	Add with extend	Dy -(Ay)	Dx -(Ax)	C	*	•	*	*
AND	AND	Logical and	<ea> Dn	Dn <ea>	–	*	*	0	0
	ANDI	And immediate	#<d>	<ea>	–	*	*	0	0
	ANDI to CCR	And immediate to CCR	#xxx	CCR					
Privileged	ANDI to SR	And immediate to SR	#xxx	SR					
—	ASL	Arithmetic shift left	Dx #<d> <ea>	Dy Dy			*		
—	ASR	Arithmetic shift right					*		
—	BCC	Branch on condition cc	<label>		–	–	–	–	–
—	BCHG	Bit test and change	Dn #<d>	<ea>	–	–		–	–
—	BCLR	Bit test and clear	Dn #<d>	<ea>	–	–		–	–
—	BRA	Branch always	<label>		–	–	–	–	–
—	BSET	Bit test and set	Dn #<d>	<ea>	–	–		–	–
—	BSR	Branch to subroutine	<label>		–	–	–	–	–
—	BTST	Bit test	Dn #<d>	<ea>	–	–		–	–
—	CHK	Check register against bounds	<ea>	Dn	–		U	U	U
—	CLR	Clear operand	<ea>		–	0	1	0	0
CMP	CMP	Compare	<ea>	Dn	–	*	*	*	*
	CMPA	Compare address	<ea>	An	–	*	*	*	*
	CMPM	Compare memory	(Ay)+	(Ax)+	–	*	*	*	*
	CMPI	Compare immediate	#<d>	<ea>	–	*	*	*	*
—	DBcc	Decrement and branch conditionally	Dn	<label>	–	–	–	–	–
—	DIVS	Signed divide	<ea>	Dn	–	*	*	*	0
—	DIVU	Unsigned divide	<ea>	Dn	–	*	*	*	0
EOR	EOR	Exclusive or	Dn	<ea>	–	*	*	0	0
	EORI	Exclusive or immediate	#<d>	<ea>	–	*	*	0	0

SIZE	NOTES.	AS OPCODES
b	(Source)$_{10}$ + (Destination)$_{10}$ → Destination	abcd
bwl	(Source) + (Destination) → Destination	addb, addw, addl
wl	(Source) + (Destination) → Destination	addw, addl
bwl	Immediate data + (Destination) → Destination	addqb, w, l
bwl	Immediate data + (Destination) → Destination	addb, w, l
bwl	(Source) + (Destination) + X → Destination	addxb, w, l
bwl	(Source) ∧ (Destination) → Destination	andb, andw, andl
bwl	Immediate data ∧ (Destination) → Destination	andb, w, l
b	(Source) ∧ CCR → CCR	
w	If supervisor, then (source) ∧ SR → SR, else TRAP	
bwl	(destination) shifted by (count) → dest N = most sig bit; CX set according to last bit shifted V set if msb changed any time during shift, else cleared	aslb, aslw, asll
bwl		asrb, asrw, asrl
bw	If (true) then PC+d → PC	bcc, bccs
bl	~(bit number) OF destination → Z (bit numb) of dest.	bchg
bl	~(bit number) OF dest → Z 0 → (bit numb) OF dest	bclr
bw	PC + d → PC	bra, bras
bl	~(bit no) of destination → Z 1 → (bit no) of destination	bset
bw	PC → -(SP); PC+d → PC	bsr, bsrs
bl	~(bit no) of destination → Z	btst
w	If D_n (N=1) < 0 or D_n (N→0) > (EA) then TRAP; else N undefined	chk
bwl	0 → destination	clrb, clrw, clrl
bwl	(Destination) - (Source)	cmpb, cmpw, cmpl
wl	(destination) - (source)	cmpw, cmpl
bwl	(destination) - (source)	cmpmb, w, l
bwl	(destination) - immediate data	cmpb, w, l
w	If false then D_n - 1 → D_n If D_n ≠ -1 then PC+d → PC else PC + 2 → PC	dbcc
w	(destination) / (source) → (destination)	divs
w	(destination) / (source) → (destination)	divu
bwl	(source) ⊕ (destination) → destination	eorb, eorw, eorl
bwl	immediate data ⊕ (destination) → destination	eorb, w, l

TYPE	VARIATION	DESCRIPTION	OPERAND TYPES 1st	OPERAND TYPES 2nd	STATUS X	N	Z	V	C
EOR cont'd	EORI to CCR	Exclusive or immediate to CCR	#xxx	CCR					
Privileged	EORI to SR	Exclusive or immediate to SR	#xxx	SR					
—	EXG	Exchange registers	Rx	Ry	-	-	-	-	-
—	EXT	Sign extend	Dn		—	*	*	0	0
—	JMP	Jump	<ea>		-	-	-	-	-
—	JSR	Jump to subroutine	<ea>		—	-	-	-	-
—	LEA	Load effective address	<ea>	An	-	-	-	-	-
—	LINK	Link and allocate	An	#<D>	-	-	-	-	-
—	LSL	Logical shift	Dx #<d>	Dy Dy		*	*	0	
—	LSR		<ea>			*	*	0	
MOVE	MOVE	Move data	<ea>	<ea>	—	*	*	0	0
	MOVEA	Move address	<ea>	An	-	-	-	-	-
Privileged	MOVEC	Move control register	Rc Rn	Rn Rc	—	-	-	-	-
	MOVEM	Move multiple registers	<ea> <list>	<list> <ea>	-	-	-	-	-
	MOVEP	Move peripheral data	Dx d(Ay)	d(Ay) Dx	—	-	-	-	-
	MOVEQ	Move quick	#<d>	Dn	—	*	*	0	0
Privilege	MOVES	Move to/from address space	Rn <ea>	<ea> Rn	-	-	-	-	—
	MOVE from SR	Move from status register	SR	<ea>	—	-	-	-	-
Privileged	MOVE to SR	Move to status register	<ea>	SR					
	MOVE from CCR	Move from condition code reg.	CCR	<ea>	—	-	-	-	—
	MOVE to CCR	Move to condition code reg	<ea>	CCR					
Privileged	MOVE USP	Move user stack pointer	USP An	An USP	-	-	-	-	-
—	MULS	Signed multiply	<ea>	Dn	-	*	*	0	0
—	MULU	Unsigned multiply	<ea>	Dn	-	*	*	0	0
—	NBCD	Negate decimal with extend	<ea>		C	U	•	U	*
NEG	NEG	Negate	<ea>		C	*	*	*	*
	NEGX	Negate with extend	<ea>		C	*	*	*	*
—	NOP	No operation	—	—	—	-	-	-	-
	NOT	Logical complement	<ea>		—	*	*	0	0

LENGTH	NOTES	AS OPCODES
b	(source) ⊕ CCR → CCR	
w	If supervisor then (source) ⊕ SR → SR else TRAP	
l	Rx ⇔ Ry	exg
wl	(dest) sign extended → dest (byte → word or word → long word)	extw, l
—	destination → PC	jmp
—	PC → -(SP), destination → PC	jsr
l	destination → An	lea
—	An → -(SP), SP → An ; SP + D → SP	link
bwl	(destination) shifted by (count) → destination	lsl b, w, l
bwl	X, C matches last bit shifted out. If count = 0, C cleared, X unaffected	lsr b, w, l
bwl	(source) → destination	movb, l), w
wl	(source) → destination	movw, l
l	If supervisor then Rc ⇄ Rn else TRAP	
wl	Register → destination, or (source) → register	movem w, l
wl	(source) → destination	movep w, l
l	immediate data → destination	moveq
bwl	If supervisor then Rn → dest (DFC) source (SFC) → Rn else TRAP	
w	SR → destination	
w	If supervisor, then (source) → SR else TRAP	
w	CCR → destination	
w	(source) → CCR	
l	If supervisor, then An → USP else TRAP	
w	(source) × (destination) → destination	muls
w		mulu
b	-(destination)$_{10}$ - X → destination	nbcd
bwl	- (destination) → destination	negb, w, l
bwl	- (destination) - X → destination	negx b, w, l
—	None.	nop
bwl	~ (destination) → destination	notb, w, l

TYPE	VARIATION	DESCRIPTION	OPERAND TYPES 1st	OPERAND TYPES 2nd	STATUS X	N	Z	V	C
OR	OR	Inclusive logical OR	<ea> Dn Dn	Dn <ea>	–	*	*	0	0
	ORI	Inclusive OR immediate	#<d>	<ea>	–	*	*	0	0
	ORI to CCR	" " "to condition codes	#xxx	CCR					
Privileged	ORI to SR	" " "to status register	#xxx	SR					
—	PEA	Push effective address	<ea>		–	–	–	–	–
Privileged	RESET	Reset external devices	—	—	–	–	–	–	–
	ROL	} Rotate without extend	Dx #<d>	Dy Dy	–	*	*	0	
	ROR		<ea>		–	*	*	0	
	ROXL	} Rotate with extend	Dx #<d>	Dy Dy		*	*	0	
	ROXR		<ea>			*	*	0	
	RTD	Return and deallocate parameters	#<d>		–	–	–	–	–
Privileged	RTE	Return from exception	—	—	–	–	–	–	–
	RTS	Return from subroutine	—	—	–	–	–	–	–
	SBCD	Subtract decimal with extend	Dx $-(A_y)$	Dy $-(A_x)$	C	U	•	U	*
	Scc	Set according to condition	<ea>		–	–	–	–	–
	STOP	Load status register and stop	#xxx						
SUB	SUB	Subtract binary	<ea> Dn Dn	Dn <ea>	C	*	*	*	*
	SUBA	Subtract address	<ea>	An	–	–	–	–	–
	SUBI	Subtract immediate	#<d>	<ea>	C	*	*	*	*
	SUBQ	Subtract quick	#<d>	<ea>	C	*	*	*	*
	SUBX	Subtract with extend	Dy $-(A_y)$	Dx $-(A_x)$	C	*	*	*	*
—	SWAP	Swap register halves	Dn		–	*	*	0	0
	TAS	Test and set	<ea>		–	*	*	0	0
	TRAP	Trap	#<v>		–	–	–	–	–
	TRAPV	Trap on overflow			–	–	–	–	–
	TST	Test	<ea>		–	*	*	0	0
	UNLK	Unlink	An		–	–	–	–	–
	RTR	Return and restore condition codes	—	—					

LENGTH	NOTES	AS OPCODES
b w l	(source) v (destination) → destination	or b, w, l
b w l	data V (destination) → destination	or b, w, l
b	(source) V CCR → CCR	
w	If supervisor then (source) VSR → SR else TRAP	
l	destination → -(SP)	pea
—	If supervisor then assert RESET line else TRAP	reset
b w l	(destination) rotated by (count) → destination	rol b, w, l
b w l	C set according to last bit shifted out, cleared for count of 0.	ror b, w, l
b w l	(destination) rotated by (count) → destination	roxl b, w, l
b w l	X C set to last bit shifted out. Count of 0 clears C, does not affect X.	roxr b, w, l
—	(SP)+ → PC ; SP+d → SP	
—	If supervisor then (SP)+ → SR ; (SP)+ → PC else trap (see RTR)	rte
—	(SP)+ → PC	rts
b	(destination)$_{10}$ - (source)$_{10}$ - X → destination	sbcd
b	If condition then 1's → destination else 0's → destination	scc
—	If supervisor then data → SR ; STOP else TRAP	stop
b w l	(destination) - (source) → destination	sub b, w, l
w l	(destination) - (source) → destination	sub w, l
b w l	(destination) - data → destination	sub b, w, l
b w l	(destination) - data → destination	subq b, w, l
b w l	(destination) - (source) - X → destination	subx b, w, l
w	Register [31:16] ↔ Register [15:0]	swap
b	(destination) tested → CC (N, Z) ; 1 → destination [7]	tas
—	PC → -(SSP), SR → -(SSP), (vector) → PC	trap
—	If V then TRAP	trapv
b w l	(destination) tested → CC (N, Z)	tst b, w, l
	An → SP ; (SP)+ → An	unlk
—	(SP)+ → CC ; (SP)+ → PC	rtr

5 CONDITION CODES.

MNEMONIC	CONDITION	DETERMINED BY
CC	carry clear	$\bar{C}$
CS	carry set	C
EQ	equal	Z
GE	$\geq$	$N.V + \bar{N}.\bar{V}$
GT	$>$	$N.V.\bar{Z} + \bar{N}.\bar{V}.\bar{Z}$
HI	high	$\bar{C}.\bar{Z}$
LE	$\leq$	$Z + N.\bar{V} + \bar{N}.V$
LS	low or same	$C + Z$
LT	$<$	$N.\bar{V} + \bar{N}.V$
MI	<0 ?	N
NE	$\neq$	$\bar{Z}$
PL	≥ 0 ?	$\bar{N}$
VC	overflow clear	$\bar{V}$
VS	overflow set	V
T	true	always true
F	false	always false

6 HIGH-LEVEL CODE IMPLEMENTATION.

6.1 FOR LOOPS.

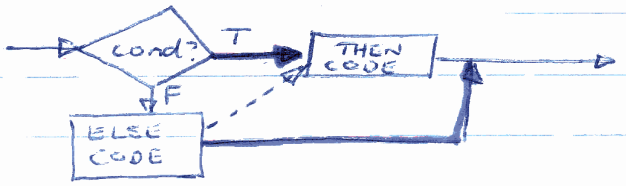
We wish to implement a loop of the form:

```
                    M DOWNTO N STEP -1
FOR I := N TO M STEP 1 DO

    BEGIN

        statements

    END.
```

This can be implemented very easily

6.2 IF - THEN - ELSE STRUCTURES

By structuring these as follows:



Where blue lines represent sequential processing and the green lines jumps. These are easily done.

6.3 CASE STATEMENTS

Consider the simple type:

```
CASE I OF
    1: statement-1
    2: statement-2
  etc.
```

this can be:

```
        movl I,d0
        addl #1,d0
        bra d0@
L1  : statement-1
L2  : statement-2
etc.
```

7. GENERAL

7.1 SOFTWARE DEVELOPMENT

As many instructions have multiple operand types, this greatly reduces the number of mnemonics as well as making the processor more flexible, powerful and consistent.

Addressing modes too are simple yet efficient. Consistency is maintained as any of the address or data registers can be used as an index register, while any address register can be used as a stack pointer.

The M68000 also allows structured modular programming as it allows code to be written in reentrant modules with easy parameter transfer, using LINK and UNLK, MOVEM. The PEA, LEA instructions, TRAP vectors and flexible return instructions also encourage structured programming.

Hardware traps have been provided to detect the following error conditions:

- word access with an odd address
- illegal instructions
- unimplemented instructions
- illegal memory access (bus error)
- divide by zero
- overflow condition code (called by TRAPV)
- register out of bounds (called by CHK)
- spurious interrupts.

Additionally the sixteen software TRAP instructions may be utilised by the programmer to provide application-oriented error detection or correction routines.

Finally, the M68000 includes a single-step trace mode, calling a trap after each instruction is executed. The user/supervisor states provide added memory and hardware protection.

7.2 DATA ORGANISATION

7.2.1 REGISTER ORGANISATION*

(a) DATA REGISTERS

Each 32-bit data register can support data operands of 1, 8, 16 or 32-bits. Byte operands occupy the low-order 8 bits, word operands the low-order 16-bits, and long-words all 32-bits. The least significant bit is addressed as bit 0; the most significant as bit 31.

When a data register is used as an operand, only the appropriate low-order part is used/changed.

(b) ADDRESS REGISTERS

Each 32-bit address register can support 32-bit addresses and 16 and 32-bits data operands. When an address register is used as the destination operand, the entire register is affected regardless of the operation size. If the operation size is word, any other operands are sign-extended to 32 bits before the operation is performed.

7.2.2 MEMORY ORGANISATION

Instructions and multibyte data are accessed only on word (even byte) boundaries, thus byte addresses are 0, 1, 2, ... FFFFFF; word addresses 0, 2, 4, ..., FFFFFE; and long-word addresses 0, 4, 8, ..., FFFFFC.

* Note a register can hold 8 digits in BCD

7.2.3 ADDRESSING

Instructions contain two types of information: the type of function to be performed and the location of the operand(s) on which to perform that function. Operand locations are specified in one of three ways:

- register specification – the register is explicitly described.
- effective address – use of the different addressing modes
- implicit reference – the definition of the instruction may imply the use of specific registers.

7.2.4 INSTRUCTION FORMAT

Instructions occupy from one to five words, these being made up of an operation word plus possible operand words or word-pairs as follows:

- operation word (specifies operation and modes)
- immediate operand (if any, 1-2 words)
- source effective address extension (if any, 1-2 words)
- destination " " (" " " ")

7.2.5 STACKS AND QUEUES

In addition to supporting arrays with the index addressing mode, the M68000 also supports stack and queue data structures with the address register postincrement and predecrement addressing modes.

The system stack is used implicitly by many instructions; user stacks and queues may be created and maintained through the addressing modes.

(a) SYSTEM STACK

Address register A7 is the system stack pointer. It is either the supervisor stack pointer (SSP) or user stack pointer (USP), depending on the state of the S bit in the status register. Each stack fills from high memory to low memory. The address mode -(SP) (AS: sp@-) creates a new item on the active stack while (SP)+ (AS: sp@+) removes an item.

The program counter is saved on the active stack on subroutine calls and restored on returns. For traps and interrupts, both the program counter and the status register are saved.

Data is always entered on a word boundary. Thus byte data is pushed or pulled from the stack in the high half of the word - the lower half is unchanged.

(b) USER STACKS

User stacks can be implemented and manipulated by employing the address register indirect with postincrement and predecrement addressing modes. Using an address register, the user may implement stacks which are filled either from high memory to low memory, or vice-versa.

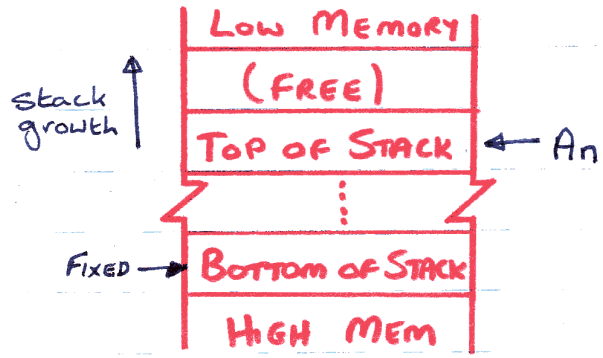
The important things to remember are:

- using predecrement, the regiate is decremented before the stack is accessed
- using postincrement, the stack is accessed before the pointe is incremented
- byte data must be put on the stack in pairs when mixed with word or long data so that the stack will not get misaligned when data is retrieved. Word and long accesses must be on word boundary (even) addresses.

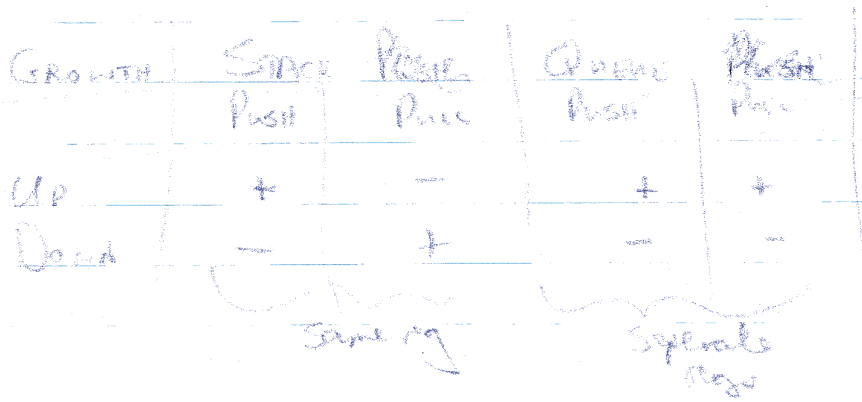
Stack growth downward in memory is implemented with:

push to -(An)
pull from (An)+

After either a push or pull operation, register An points to the last (top) item on the stack:



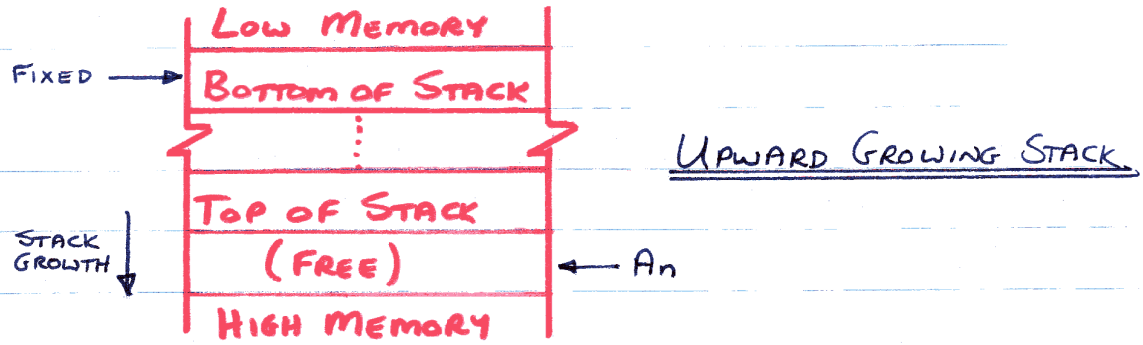
DOWNWARD GROWING STACK



Stack growth upward in memory is implemented with:

push to (An)+
pull from -(An)

After each push or pull operation, An points to the next free space on the stack:



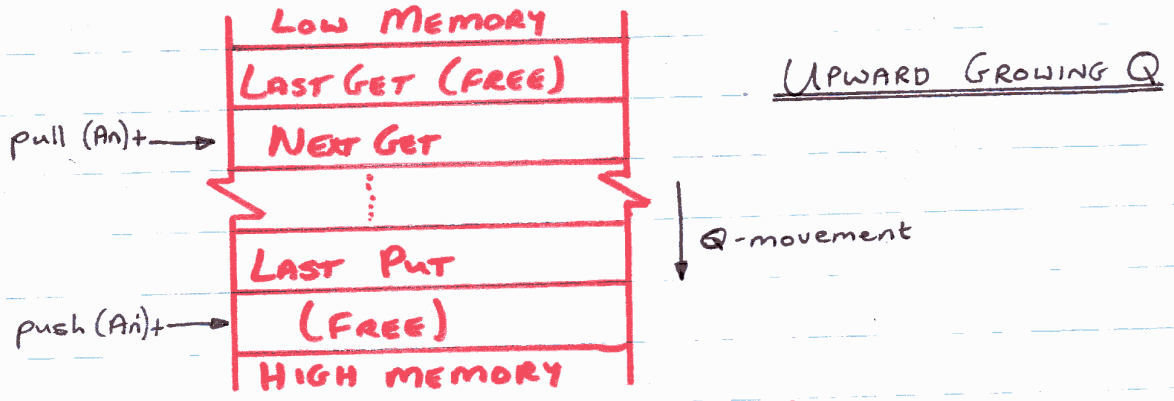
(c) QUEUES

Using a pair of address registers, the user may implement queues which are filled either from high memory to low memory, or vice-versa. Because queues are pushed from one end and pulled from the other, two registers are required, the push and pull pointers.

An upward growing queue is implemented with:
push (Apush)+
pull (Apull)+

After a push operation, the push index points to the next available space in the queue and the unchanged pull index points to the next item to remove. After a pull operation, the pull pointer points to points to the next item

to be removed while the unchanged push index points to the next available space:



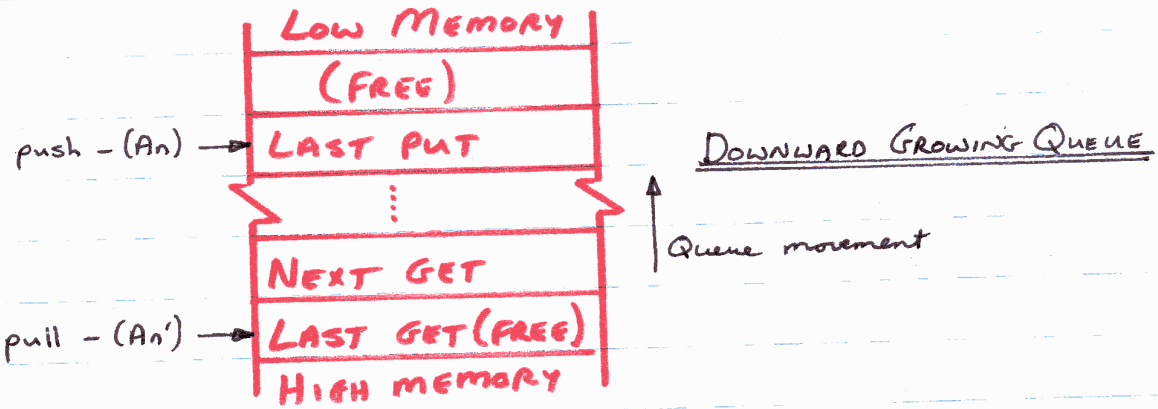
If this is to be implemented as a circular buffer, the address register should be checked, and if necessary, adjusted before the put or get operation is performed. The address register is adjusted by subtracting the buffer length in bytes.

A downward growing queue is implemented with:

push - (An)

pull - (An')

After a push or pull, the pull index points to the last item removed from the queue, while the push index points to the last item pushed into the queue:



If the queue is to be implemented as a circular buffer, the operation should first be performed and then the pointer checked, and if necessary, adjusted by adding the buffer length in bytes.

7.3 I/O ON THE NCR TOWER 1632.

I/O on the Tower is done by means of O/S routines called by TRAP #0. d0 is used to hold a number specifying the requested routine, the available options being:

0 - access (33)
1 - acct (51)
2 - alarm (27)
3 - brk (17)
4 - chdir (12)
5 - chmod (15)
6 - chown (16)
7 - close (6)
8 - creat (8)
9 - dup (41)
10 - exec (11)
11 - exit (1)
12 - fork (2)
13 - getpid (20)
14 - getuid (24)
15 - getgid (47)
16 - ioctl (54)
17 - stty (31)
18 - gtty (32)
19 - kill (37)
20 - link (9)
21 - lock (53)
22 - locking (45)
23 - lseek (19)
24 - mknod (14)
25 - mount (21)
26 - umount (22)
27 - nice (34)
28 - open (5)
29 - pause (29)
30 - phys (52)
31 - pipe (42)
32 - profil (44)
33 - ptrace (26)
34 - pwrnote, pwrtime (57)
35 - read (3)
36 - setuid (23)
37 - setgid (46)
38 - signal (48)
39 - stat (18)
40 - fstat (28)
41 - stime (25)
42 - sync (36)
43 - ftime (35)
44 - time (13)
45 - times (43)
46 - umask (60)
47 - unlink (10)
48 - utime (30)
49 - wait (7)
50 - write (4)

For read (3) and write (4), the following parameters are required :

a0 - file descriptor
d1 - address of the start of the I/O buffer
a1 - no of bytes to be read/written.

The number of bytes read/written is returned in d0, and the carry bit cleared on success. File descriptors for the standard files are :

0 - standard input
1 - standard output
2 - error file