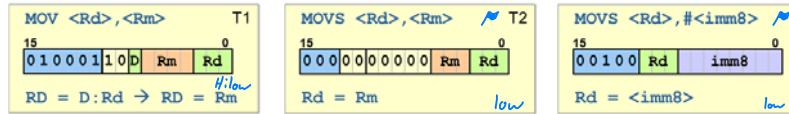
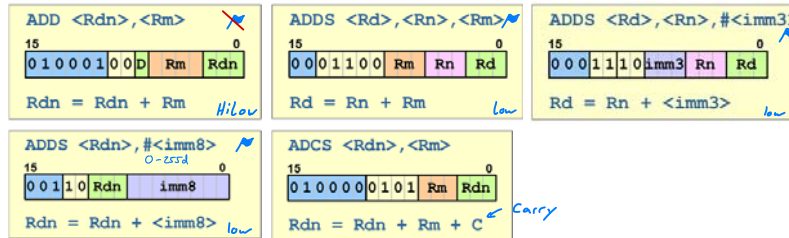


ARM v6-M Instruction Set

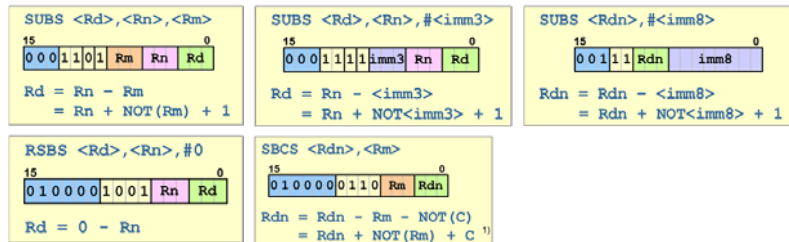
MOV



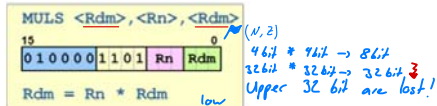
ADD



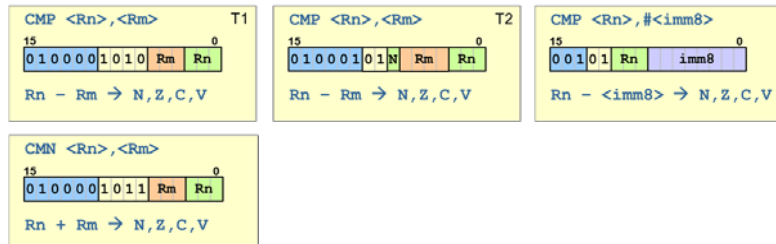
Subtract



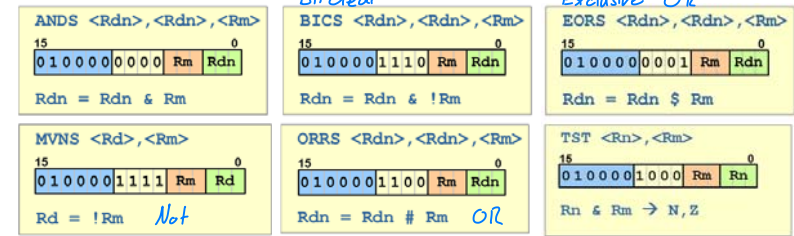
Multiply



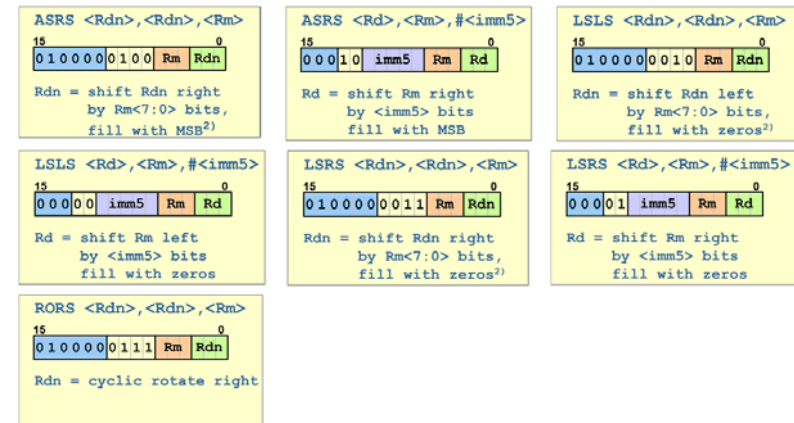
Compare



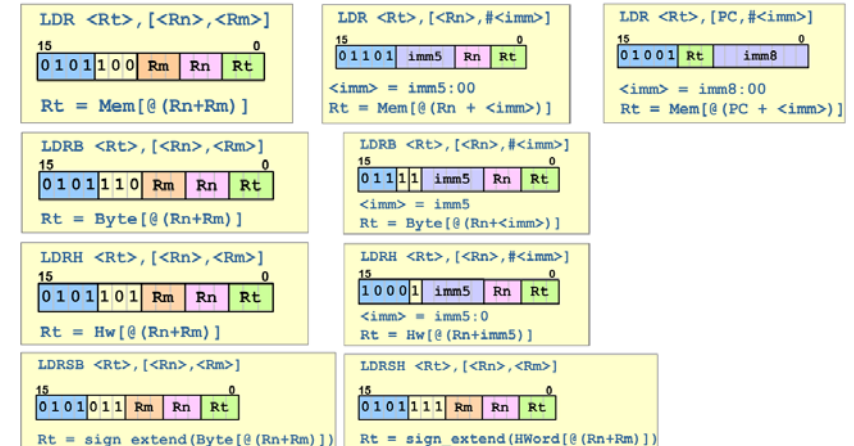
Logical (update N and Z flags)



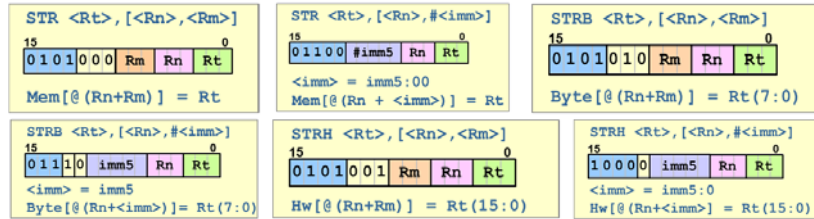
Shift/Rotate



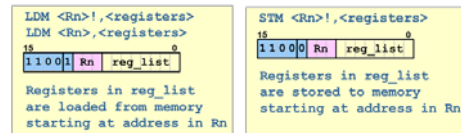
Load



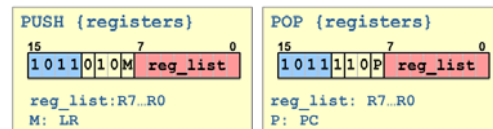
Store



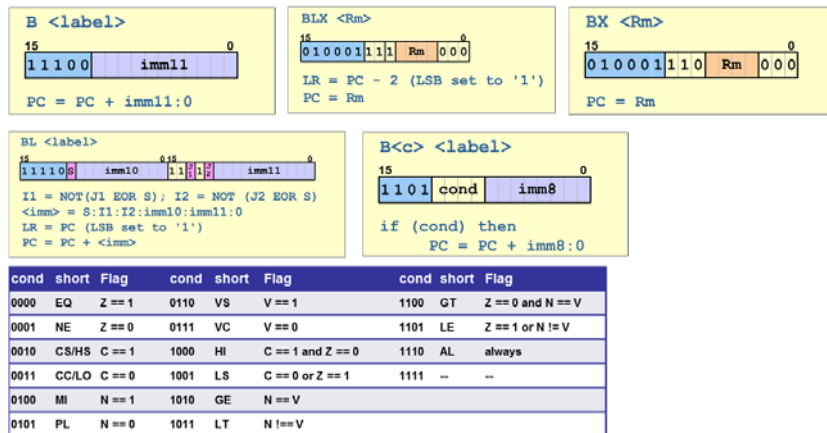
Load/Store Multiple



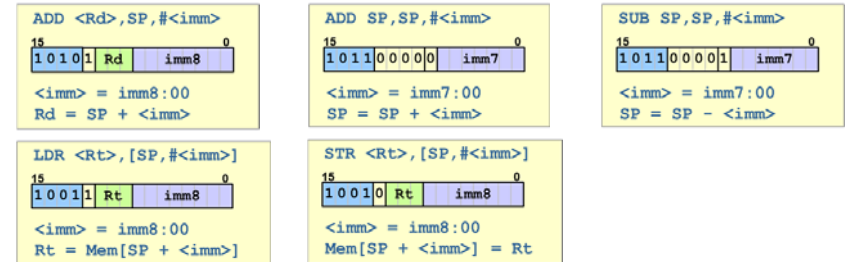
Push/Pop



Branch



Stack Operations



Extend



Pseudo Instructions

LDR <Rt>, <label> => LDR <Rt>, [PC, #<imm>]
 LDR <Rt>, =<value> => LDR <Rt>, [PC, #<imm>]
 ...
 Literalpool
 DCD value

Weitere Befehle

REV REV16 REVSH SVC CPSID CPSIE SETEND BKPT NOP SEV
 WFE WFI YIELD

Thumb® 16-bit Instruction Set
Quick Reference Card

This card lists all Thumb instructions available on Thumb-capable processors earlier than ARM®v6T2. In addition, it lists all Thumb-2 16-bit instructions. The instructions shown on this card are all 16-bit in Thumb-2, except where noted otherwise. All registers are Lo (R0-R7) except where specified. Hi registers are R8-R15.

Key to Tables				
§	See Table ARM architecture versions .		<loreglist+LR>	A comma-separated list of Lo registers, plus the LR, enclosed in braces, { and }.
<loreglist>	A comma-separated list of Lo registers, enclosed in braces, { and }.		<loreglist+PC>	A comma-separated list of Lo registers, plus the PC, enclosed in braces, { and }.

Operation		§	Assembler	Updates	Action	Notes
Move	Immediate	6	MOVS Rd, #<imm>	N Z	Rd := imm	imm range 0-255.
	Lo to Lo		MOVS Rd, Rm	N Z	Rd := Rm	Synonym of LSLS Rd, Rm, #0
	Hi to Lo, Lo to Hi, Hi to Hi		MOV Rd, Rm		Rd := Rm	Not Lo to Lo.
	Any to Any		MOV Rd, Rm		Rd := Rm	Any register to any register.
Add	Immediate 3	T2	ADDS Rd, Rn, #<imm>	N Z C V	Rd := Rn + imm	imm range 0-7.
	All registers Lo		ADDS Rd, Rn, Rm	N Z C V	Rd := Rn + Rm	
	Hi to Lo, Lo to Hi, Hi to Hi		ADD Rd, Rd, Rm		Rd := Rd + Rm	Not Lo to Lo.
	Any to Any		ADD Rd, Rd, Rm		Rd := Rd + Rm	Any register to any register.
	Immediate 8		ADDS Rd, Rd, #<imm>	N Z C V	Rd := Rd + imm	imm range 0-255.
	With carry		ADCS Rd, Rd, Rm	N Z C V	Rd := Rd + Rm + C-bit	
	Value to SP		ADD SP, SP, #<imm>		SP := SP + imm	imm range 0-508 (word-aligned).
	Form address from SP		ADD Rd, SP, #<imm>		Rd := SP + imm	imm range 0-1020 (word-aligned).
	Form address from PC		ADR Rd, <label>		Rd := label	label range PC to PC+1020 (word-aligned).
Subtract	Lo and Lo		SUBS Rd, Rn, Rm	N Z C V	Rd := Rn – Rm	
	Immediate 3		SUBS Rd, Rn, #<imm>	N Z C V	Rd := Rn – imm	imm range 0-7.
	Immediate 8		SUBS Rd, Rd, #<imm>	N Z C V	Rd := Rd – imm	imm range 0-255.
	With carry		SBCS Rd, Rd, Rm	N Z C V	Rd := Rd – Rm – NOT C-bit	
	Value from SP		SUB SP, SP, #<imm>		SP := SP – imm	imm range 0-508 (word-aligned).
Multiply	Negate		RSBS Rd, Rn, #0	N Z C V	Rd := – Rn	Synonym: NEGS Rd, Rn
Multiply	Multiply		MULS Rd, Rm, Rd	N Z * *	Rd := Rm * Rd	* C and V flags unpredictable in §4T, unchanged in §5T and above
Compare			CMP Rn, Rm	N Z C V	update APSR flags on Rn – Rm	Can be Lo to Lo, Lo to Hi, Hi to Lo, or Hi to Hi.
	Negative		CMN Rn, Rm	N Z C V	update APSR flags on Rn + Rm	
	Immediate		CMP Rn, #<imm>	N Z C V	update APSR flags on Rn – imm	imm range 0-255.
Logical	AND		ANDS Rd, Rd, Rm	N Z	Rd := Rd AND Rm	
	Exclusive OR		EORS Rd, Rd, Rm	N Z	Rd := Rd EOR Rm	
	OR		ORRS Rd, Rd, Rm	N Z	Rd := Rd OR Rm	
	Bit clear		BICS Rd, Rd, Rm	N Z	Rd := Rd AND NOT Rm	
	Move NOT		MVNS Rd, Rm	N Z	Rd := NOT Rm	
	Test bits		TST Rn, Rm	N Z	update APSR flags on Rn AND Rm	
Shift/rotate	Logical shift left		LSLS Rd, Rm, #<shift>	N Z C*	Rd := Rm << shift	Allowed shifts 0-31. * C flag unaffected if shift is 0.
			LSLS Rd, Rd, Rs	N Z C*	Rd := Rd << Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Logical shift right		LSRS Rd, Rm, #<shift>	N Z C	Rd := Rm >> shift	Allowed shifts 1-32.
			LSRS Rd, Rd, Rs	N Z C*	Rd := Rd >> Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Arithmetic shift right		ASRS Rd, Rm, #<shift>	N Z C	Rd := Rm ASR shift	Allowed shifts 1-32.
			ASRS Rd, Rd, Rs	N Z C*	Rd := Rd ASR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Rotate right		RORS Rd, Rd, Rs	N Z C*	Rd := Rd ROR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.

Thumb 16-bit Instruction Set

Quick Reference Card

Operation		§	Assembler	Action	Notes
Load	with immediate offset, word		LDR Rd, [Rn, #<imm>]	Rd := [Rn + imm]	imm range 0-124, multiple of 4.
	halfword		LDRH Rd, [Rn, #<imm>]	Rd := ZeroExtend([Rn + imm][15:0])	Clears bits 31:16. imm range 0-62, even.
	byte		LDRB Rd, [Rn, #<imm>]	Rd := ZeroExtend([Rn + imm][7:0])	Clears bits 31:8. imm range 0-31.
	with register offset, word		LDR Rd, [Rn, Rm]	Rd := [Rn + Rm]	
	halfword		LDRH Rd, [Rn, Rm]	Rd := ZeroExtend([Rn + Rm][15:0])	Clears bits 31:16
	signed halfword		LDRSH Rd, [Rn, Rm]	Rd := SignExtend([Rn + Rm][15:0])	Sets bits 31:16 to bit 15
	byte		LDRB Rd, [Rn, Rm]	Rd := ZeroExtend([Rn + Rm][7:0])	Clears bits 31:8
	signed byte		LDRSB Rd, [Rn, Rm]	Rd := SignExtend([Rn + Rm][7:0])	Sets bits 31:8 to bit 7
	PC-relative		LDR Rd, <label>	Rd := [label]	label range PC to PC+1020 (word-aligned).
	SP-relative		LDR Rd, [SP, #<imm>]	Rd := [SP + imm]	imm range 0-1020, multiple of 4.
	Multiple, not including base		LDM Rn!, <loreglist>	Loads list of registers (not including Rn)	Always updates base register, Increment After.
	Multiple, including base		LDM Rn, <loreglist>	Loads list of registers (including Rn)	Never updates base register, Increment After.
Store	with immediate offset, word		STR Rd, [Rn, #<imm>]	[Rn + imm] := Rd	imm range 0-124, multiple of 4.
	halfword		STRH Rd, [Rn, #<imm>]	[Rn + imm][15:0] := Rd[15:0]	Ignores Rd[31:16]. imm range 0-62, even.
	byte		STRB Rd, [Rn, #<imm>]	[Rn + imm][7:0] := Rd[7:0]	Ignores Rd[31:8]. imm range 0-31.
	with register offset, word		STR Rd, [Rn, Rm]	[Rn + Rm] := Rd	
	halfword		STRH Rd, [Rn, Rm]	[Rn + Rm][15:0] := Rd[15:0]	Ignores Rd[31:16]
	byte		STRB Rd, [Rn, Rm]	[Rn + Rm][7:0] := Rd[7:0]	Ignores Rd[31:8]
	SP-relative, word		STR Rd, [SP, #<imm>]	[SP + imm] := Rd	imm range 0-1020, multiple of 4.
	Multiple		STM Rn!, <loreglist>	Stores list of registers	Always updates base register, Increment After.
Push	Push		PUSH <loreglist>	Push registers onto full descending stack	
	Push with link		PUSH <loreglist>+LR	Push LR and registers onto full descending stack	
Pop	Pop		POP <loreglist>	Pop registers from full descending stack	
	Pop and return	4T	POP <loreglist>+PC	Pop registers, branch to address loaded to PC	
	Pop and return with exchange	5T	POP <loreglist>+PC	Pop, branch, and change to ARM state if address[0] = 0	
If-Then	If-Then	T2	IT{pattern} {cond}	Makes up to four following instructions conditional, according to pattern. pattern is a string of up to three letters. Each letter can be T (Then) or E (Else).	The first instruction after IT has condition cond. The following instructions have condition cond if the corresponding letter is T, or the inverse of cond if the corresponding letter is E. See Table Condition Field .
Branch	Conditional branch		B{cond} <label>	If {cond} then PC := label	label must be within – 252 to + 258 bytes of current instruction. See Table Condition Field .
	Compare, branch if (non) zero	T2	CB{N}Z Rn, <label>	If Rn {== !=} 0 then PC := label	label must be within +4 to +130 bytes of current instruction.
	Unconditional branch		B <label>	PC := label	label must be within ±2KB of current instruction.
	Long branch with link		BL <label>	LR := address of next instruction, PC := label	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch and exchange		BX Rm	PC := Rm AND 0xFFFFFFF	Change to ARM state if Rm[0] = 0.
	Branch with link and exchange	5T	BLX <label>	LR := address of next instruction, PC := label Change to ARM	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch with link and exchange	5T	BLX Rm	LR := address of next instruction, PC := Rm AND 0xFFFFFFF	Change to ARM state if Rm[0] = 0.
Extend	Signed, halfword to word	6	SXTH Rd, Rm	Rd[31:0] := SignExtend(Rm[15:0])	
	Signed, byte to word	6	SXTB Rd, Rm	Rd[31:0] := SignExtend(Rm[7:0])	
	Unsigned, halfword to word	6	UXTH Rd, Rm	Rd[31:0] := ZeroExtend(Rm[15:0])	
	Unsigned, byte to word	6	UTB Rd, Rm	Rd[31:0] := ZeroExtend(Rm[7:0])	
Reverse	Bytes in word	6	REV Rd, Rm	Rd[31:24] := Rm[7:0], Rd[23:16] := Rm[15:8], Rd[15:8] := Rm[23:16], Rd[7:0] := Rm[31:24]	
	Bytes in both halfwords	6	REV16 Rd, Rm	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:24] := Rm[23:16], Rd[23:16] := Rm[31:24]	
	Bytes in low halfword, sign extend	6	REVSH Rd, Rm	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:16] := Rm[7] * &FFFF	

Thumb 16-bit Instruction Set

Quick Reference Card

Operation		§	Assembler	Action	Notes
Processor state change	Supervisor Call		SVC <immed_8>	Supervisor Call processor exception	8-bit immediate value encoded in instruction. Formerly SWI.
	Change processor state	6	CPSID <iflags>	Disable specified interrupts	
		6	CPSIE <iflags>	Enable specified interrupts	
	Set endianness	6	SETEND <endianness>	Sets endianness for loads and saves.	
	Breakpoint	5T	BKPT <immed_8>	Prefetch abort <i>or</i> enter debug state	<endianness> can be BE (Big Endian) or LE (Little Endian). 8-bit immediate value encoded in instruction.
No Op	No operation		NOP	None, might not even consume any time.	Real NOP available in ARM v6K and above.
Hint	Set event	T2	SEV	Signal event in multiprocessor system.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Wait for event	T2	WFE	Wait for event, IRQ, FIQ, Imprecise abort, or Debug entry request.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Wait for interrupt	T2	WFI	Wait for IRQ, FIQ, Imprecise abort, or Debug entry request.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Yield	T2	YIELD	Yield control to alternative thread.	Executes as NOP in Thumb-2. Functionally available in ARM v7.

Condition Field	
Mnemonic	Description
EQ	Equal
NE	Not equal
CS / HS	Carry Set / Unsigned higher or same
CC / LO	Carry Clear / Unsigned lower
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always. Do not use in B{cond}

In Thumb code for processors earlier than ARMv6T2, cond must not appear anywhere except in Conditional Branch (B{cond}) instructions.

In Thumb-2 code, cond can appear in any of these instructions (except CBZ, CBNZ, CPSID, CPSIE, IT, and SETEND).
The condition is encoded in a preceding IT instruction (except in the case of B{cond} instructions).
If IT instructions are explicitly provided in the Assembly language source file, the conditions in the instructions must match the corresponding IT instructions.

ARM architecture versions	
4T	All Thumb versions of ARM v4 and above.
5T	All Thumb versions of ARM v5 and above.
6	All Thumb versions of ARM v6 and above.
T2	All Thumb-2 versions of ARM v6 and above.

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This reference card is intended only to assist the reader in the use of the product. ARM Ltd shall not be liable for any loss or damage arising from the use of any information in this reference card, or any error or omission in such information, or any incorrect use of the product.

Document Number

ARM QRC 0006E

Change Log

Issue	Date	Change
A	Nov 2004	First Release
B	May 2005	RVCT 2.2 SP1
C	March 2006	RVCT 3.0
D	March 2007	RVCT 3.1
E	Sept 2008	RVCT 4.0

ARM® Thumb® Cortex-M0/M1 Instruction Set ordered by machine code

This card lists all Thumb instructions ordered by machine code to ease manually disassemble Thumb code.

See the respective *Thumb® 16-bit Instruction Set Quick Reference Card* for details on the individual instructions.

Version 1.3, 2019-08-20, Andreas Gieriet

0000 - 0x0xxx Instructions			
0000 0000 00mm mddd	MOVS Rddd, Rmmm ; Rddd = Rmmm	--> alias for LSLS Rddd,Rmmm,#0	
0000 0iii iimm mddd	LSLS Rddd, Rmmm, #0biiiii; Rddd = Rmmm LSL #0b0iiii		
0000 1iii iimm mddd	LSRS Rddd, Rmmm, #0biiiii; Rddd = Rmmm LSR #0b0iiii		
0001 - 0x1xxx Instructions			
0001 0iii iimm mddd	ASRS Rddd, Rmmm, #0biiiii; Rddd = Rmmm ASR #0b0iiii		
0001 100m mnnn nddd	ADDS Rddd, Rnnn, Rmmm ; Rddd = Rnnn + Rmmm		
0001 101m mnnn nddd	SUBS Rddd, Rnnn, Rmmm ; Rddd = Rnnn - Rmmm		
0001 110i iinn nddd	ADDS Rddd, Rnnn, #0biii ; Rddd = Rnnn + #0b0iii		
0001 111i iinn nddd	SUBS Rddd, Rnnn, #0biii ; Rddd = Rnnn - #0b0iii		
0010 - 0x2xxx Instructions			
0010 0ddd iiiii iiiii	MOVS Rddd, #0biiiii; Rddd = #0b0iiiiiii		
0010 1nnn iiiii iiiii	CMP Rnnn, #0biiiii; flags = Rnnn - #0b0iiiiiii		
0011 - 0x3xxx Instructions			
0011 0ddd iiiii iiiii	ADDS Rddd, #0biiiii; Rddd = Rddd + #0b0iiiiiii		
0011 1ddd iiiii iiiii	SUBS Rddd, #0biiiii; Rddd = Rddd - #0b0iiiiiii		
0100 - 0x4xxx Instructions			
0100 0000 00mm mddd	ANDS Rddd, Rmmm ; Rddd = Rddd & Rmmm		
0100 0000 01mm mddd	EORS Rddd, Rmmm ; Rddd = Rddd ^ Rmmm		
0100 0000 10mm mddd	LSLS Rddd, Rmmm ; Rddd = Rddd LSL Rmmm		
0100 0000 11mm mddd	LSRS Rddd, Rmmm ; Rddd = Rddd LSR Rmmm		
0100 0001 00mm mddd	ASRS Rddd, Rmmm ; Rddd = Rddd ASR Rmmm		
0100 0001 01mm mddd	ADCS Rddd, Rmmm ; Rddd = Rddd + Rmmm + carry		
0100 0001 10mm mddd	SBCS Rddd, Rmmm ; Rddd = Rddd - Rmmm - ~carry		
0100 0001 11mm mddd	RORS Rddd, Rmmm ; Rddd = Rddd ROR Rmmm		
0100 0010 00mm mddd	TST Rddd, Rmmm ; flags : Rddd & Rmmm		
0100 0010 01mm mddd	RSBS Rddd, Rmmm, #0 ; Rddd = 0 - Rmmm --> alias for NEGS Rddd, Rmmm		
0100 0010 10mm mnnn	CMP Rnnn, Rmmm ; flags : Rnnn - Rmmm		
0100 0010 11mm mnnn	CMN Rnnn, Rmmm ; flags : Rnnn + Rmmm		
0100 0011 00mm mddd	ORRS Rddd, Rmmm ; Rddd = Rddd Rmmm		
0100 0011 01mm mddd	MULS Rddd, Rmmm, Rddd ; Rddd = Rddd * Rmmm		
0100 0011 10mm mddd	BICS Rddd, Rmmm ; Rddd = Rddd & ~Rmmm --> bit clear		
0100 0011 11mm mddd	MVNS Rddd, Rmmm ; Rddd = ~Rmmm		
0100 0100 0mm mddd	ADD Rddd, Rmmm ; Rddd = Rddd + Rmmm		
0100 0101 0mm mnnn	CMP Rnnn, Rmmm ; flags : Rnnn - Rmmm		
0100 0110 0mm mddd	MOV Rddd, Rmmm ; Rddd = Rmmm		
0100 0111 0mm m...	BX Rmmm ; PC=Rmmm (mnnn=0b1111: unpredictable)		
0100 0111 1mm m...	BLX Rmmm ; LR = IPC+2, PC=Rmmm (mnnn=0b1111: unpredictable)		
0100 1ttt iiiii iiiii	LDR Rttt, [PC, #off] ; Rttt = [(IPC+4)&~0b011]+0b0iiiiiii00 --> +1020 max		
	LDR Rttt, label ; --> the assembler calculates the above from the label		
	LDR Rttt, =lab ; --> pseudo instruction: the assembler stores the lab/lit		
	LDR Rttt, =lit ; in litpool, access PC relative with LDR Rttt,litpool		
0101 - 0x5xxx Instructions			
0101 000m mnnn nttt	STR Rttt, [Rnnn, Rmmm] ; [Rnnn + Rmmm] = Rttt		
0101 001m mnnn nttt	STRH Rttt, [Rnnn, Rmmm] ; [Rnnn + Rmmm] = Rttt --> low half		
0101 010m mnnn nttt	STRB Rttt, [Rnnn, Rmmm] ; [Rnnn + Rmmm] = Rttt --> low byte		
0101 011m mnnn nttt	LDRSB Rttt, [Rnnn, Rmmm] ; Rttt<sssl> = [Rnnn + Rmmm]<1> --> low byte		
0101 100m mnnn nttt	LDR Rttt, [Rnnn, Rmmm] ; Rttt = [Rnnn + Rmmm]		
0101 101m mnnn nttt	LDRH Rttt, [Rnnn, Rmmm] ; Rttt<0021> = [Rnnn + Rmmm]<21> --> low half		
0101 110m mnnn nttt	LDRB Rttt, [Rnnn, Rmmm] ; Rttt<0001> = [Rnnn + Rmmm]<1> --> low byte		
0101 111m mnnn nttt	LDRSH Rttt, [Rnnn, Rmmm] ; Rttt<ss21> = [Rnnn + Rmmm]<21> --> low half		
0110 - 0x6xxx Instructions			
0110 0iii iinn nttt	STR Rttt, [Rnnn, #off] ; [Rnnn + 0b0iiii00] = Rttt --> +124 max		
0110 1iii iinn nttt	LDR Rttt, [Rnnn, #off] ; Rttt = [Rnnn + 0x0iiii00] --> +124 max		
0111 - 0x7xxx Instructions			
0111 0iii iinn nttt	STRB Rttt, [Rnnn, #off] ; [Rnnn + 0b0iiii] = Rttt --> +31 max, low byte		
0111 1iii iinn nttt	LDRB Rttt, [Rnnn, #off] ; Rttt<0001> = [Rnnn + 0x0iiii]<1> --> +31 max, low byte		
1000 - 0x8xxx Instructions			
1000 0iii iinn nttt	STRH Rttt, [Rnnn, #off] ; [Rnnn + 0x0iiii0] = Rttt --> +62 max, low half		
1000 1iii iinn nttt	LDRH Rttt, [Rnnn, #off] ; Rttt<0021> = [Rnnn + 0x0iiii0]<21> --> +62 max, low half		
1001 - 0x9xxx Instructions			
1001 0ttt iiiii iiiii	STR Rttt, [SP, #off] ; [SP + 0b0iiiiiii00] = Rttt --> +1020 max		
1001 1ttt iiiii iiiii	LDR Rttt, [SP, #off] ; Rttt = [SP + 0b0iiiiiii00] --> +1020 max		

- 1) IPC is the PC of the current instruction (IPC+4 is given by the pipeline, IPC+2/+4 is the return address in the LR)
- 2) a dot means don't care, but must be set to 0.
- 3) <321>: word, <21>: low half word, <1>: low byte, <0001>: zero extend byte, <sssl>: sign extend byte, etc.
- 4) Undefined instructions can be used to emulate instructions (they trigger the undefined exception).

1010 - 0xAxxx Instructions			
1010 0ddd iiiii iiiii	ADR Rddd, label ; Rddd = ((IPC+4)&~0b011)+0b0iiiiiii00 --> +1020 max		
1010 1ddd iiiii iiiii	ADD Rddd, SP, #off ; Rddd = SP + 0b0iiiiiii00 --> +1020 max		
1011 - 0xBxxx Instructions			
1011 0000 0iii iiiii	ADD SP, SP, #off ; SP = SP + 0b0iiiiiii00 --> +508 max		
1011 0000 1iii iiiii	SUB SP, SP, #off ; SP = SP - 0b0iiiiiii00 --> +508 max		
1011 0001 0iii iiiii	CBZ Rnnn, label ; if Rnnn==zero, PC = IPC+4 + 0x0iiiiii0 --> +126 max		
1011 0010 00mm mddd	SXTH Rddd, Rmmm ; Rddd<ss21> = Rmmm<4321> --> low half		
1011 0010 01mm mddd	SXTB Rddd, Rmmm ; Rddd<sssl> = Rmmm<4321> --> low byte		
1011 0010 10mm mddd	UXTH Rddd, Rmmm ; Rddd<0021> = Rmmm<4321> --> low half		
1011 0010 11mm mddd	UXTB Rddd, Rmmm ; Rddd<0001> = Rmmm<4321> --> low byte		
1011 0100 rrrr rrrr	PUSH {reg0-7} ; rrrrrrrr = Lo reg-mask --> pushes regs to SP (decrements SP)		
1011 0101 rrrr rrrr	PUSH {LR,reg0-7} ; rrrrrrrr = Lo reg-mask --> pushes regs to SP (decrements SP)		
1011 0110 0100 xxxx	- ; unpredictable		
1011 0110 0101 0...	SETEND LE ; sets little-endian mode in CPSR		
1011 0110 0101 1...	SETEND BE ; sets big-endian mode in CPSR		
1011 0110 0110 0aif	CPSIE aif ; Enable Processor State --> a=imprecise-abort, i=IRQ, f=FIQ		
1011 0110 0111 0aif	CPSID aif ; Disable Processor State --> a=imprecise-abort, i=IRQ, f=FIQ		
1011 0110 011x 1xxx	- ; unpredictable		
1011 1011 0iii iiiii	CBNZ Rnnn, label ; if Rnnn!=zero, PC = IPC+4 + 0x0iiiiii0 --> +126 max		
1011 1010 00mm mddd	REV Rddd, Rmmm ; Rddd<4321> = Rmmm<1234> --> reverse all		
1011 1010 01mm mddd	REV16 Rddd, Rmmm ; Rddd<4321> = Rmmm<3412> --> reverse low half, rev. upper half		
1011 1010 10xx xxxx	- ; undefined		
1011 1010 11mm mddd	REVSH Rddd, Rmmm ; Rddd<4321> = Rmmm<ss12> --> reverse low half, sign extended		
1011 1100 rrrr rrrr	POP {reg0-7} ; rrrrrrrr = Lo reg-mask --> pops regs from SP (increments SP)		
1011 1101 rrrr rrrr	POP {PC,reg0-7} ; rrrrrrrr = Lo reg-mask --> pops regs from SP (increments SP)		
1011 1110 iiiii iiiii	BKPT #0biiiii; ; breakpoint, arg ignored by HW		
1011 1111 0000 0000	NOP ; do nothing		
1011 1111 0001 0000	YIELD ; do nothing, NOP-Hint: signal to HW to suspend/resume threads		
1011 1111 0010 0000	WFE ; do nothing, NOP-Hint, wait for event		
1011 1111 0011 0000	WFI ; do nothing, NOP-Hint: wait for interrupt		
1011 1111 0100 0000	SEV ; do nothing, NOP-Hint: signal event to multi-processor system		
1011 1111 eeee eeee	ITeol eend ; if-then- sel-mnnn: T-then/E-else, eend=eeee: as for Bcc<11>8>		
1100 - 0xCxxx Instructions			
1100 0nnn rrrr rrrr	STMIA Rnnn! {reg0-7} ; rrrrrrrr = Lo reg-mask, inc Rnnn		
1100 1nnn rrrr rrrr	LDmia Rnnn! {reg0-7} ; rrrrrrrr = Lo reg-mask, inc Rnnn if Rnnn not in mask		
	LDmia Rnnn {reg0-7} ; rrrrrrrr = Lo reg-mask, load Rnnn if Rnnn in mask		
1101 - 0xDxxx Instructions			
1101 0000 iiiii iiiii	BEQ label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 0001 iiiii iiiii	BNE label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 0010 iiiii iiiii	BHS/BCS label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 0011 iiiii iiiii	BLO/BCC label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 0100 iiiii iiiii	BPL label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 0101 iiiii iiiii	BMI label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 0110 iiiii iiiii	BVS label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 0111 iiiii iiiii	BVC label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 1000 iiiii iiiii	BHI label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 1001 iiiii iiiii	BLS label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 1010 iiiii iiiii	BGE label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 1011 iiiii iiiii	BLT label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 1100 iiiii iiiii	BGT label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 1101 iiiii iiiii	BLE label ; if true, PC = IPC+4 + 0biiiii00 --> -256/+254 max		
1101 1110 xxxx xxxx	- ; undefined --> can be used for instruction emulation		
1101 1111 iiiii iiiii	SVC #0biiiii; ; supervisor call (formerly called SWI), arg ignored by HW		
1110 - 0xExxx Instructions			
1110 0iii iiiii iiiii	B label ; PC = IPC+4 + 0biiiii00 --> -2048/+2046 max		
1110 1xxx xxxx xxxx	- ; 32-bit instructions		
1111 - 0xFxxx Instructions			
1111 0iii iiiii iiiii 11y1 ziii iiiii iiiii	BL label ; LR=IPC+4,PC=IPC+4+0bXYZiii...ii0,X,Y,Z=f(x,y,z), +/-16M		
1111 0011 1110 1111 1000 dddd ssss ssss	MRS Rddd,S; Rddd = special register S (encoded as 0bssssssss)		
1111 0011 1000 mnnn 1000 1000 ssss ssss	MSR S,Rmmm; special register S (encoded as 0bssssssss) = Rmmm		
1111 0011 1011 1111 1000 1111 0100 1111	DSB ; data synchronization barrier		
1111 0011 1011 1111 1000 1111 0101 1111	DMB ; data memory barrier		
1111 0011 1011 1111 1000 1111 0110 1111	ISB ; instruction synchronization barrier		
1111 1xxx xxxx xxxx xxxx xxxx xxxx xxxx	- ; other 32-bit instructions		

- 5) Unpredictable instructions do any unpredictable actions and are therefore illegal instructions.
- 6) Unallocated codes are undefined unless they are explicitly marked as unpredictable.
- 7) CBZ, CBNZ, IT are the only 16 bit instructions which are not part of Cortex-M0/M1 Thumb code.
- 8) BL, DMB, DSB, ISB, MRS, MSR are the only 32 bit instructions as part of the Cortex-M0/M1 instruction set.

Flag- Dependent

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
CS	Carry set	C == 1
CC	Carry clear	C == 0
MI	Minus/negative	N == 1
PL	Plus/positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0

Arithmetic - unsigned: higher and lower

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
HS (=CS)	Unsigned higher or same	C == 1
LO (=CC)	Unsigned lower	C == 0
HI	Unsigned higher	C == 1 and Z == 0
LS	Unsigned lower or same	C == 0 or Z == 1

Arithmetic - signed: greater and less

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
MI	Minus/negative	N == 1
PL	Plus/positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	Z == 0 and N == V
LE	Signed less than or equal	Z == 1 or N != V

C Reference Card (ANSI)

Program Structure/Functions

<i>type fnc</i> (<i>type</i> ₁ , ...);	function prototype
<i>type name</i> ;	variable declaration
int main(void) {	main routine
<i>declarations</i>	local variable declarations
<i>statements</i>	
}	
<i>type fnc</i> (<i>arg</i> ₁ , ...) {	function definition
<i>declarations</i>	local variable declarations
<i>statements</i>	
return <i>value</i> ;	
}	
/* */	comments
int main(int argc, char *argv[])	main with args
exit(<i>arg</i>);	terminate execution

C Preprocessor

include library file	#include <filename>
include user file	#include "filename"
replacement text	#define <i>name text</i>
replacement macro	#define <i>name</i> (<i>var</i>) <i>text</i>
Example. #define max(A,B) ((A)>(B) ? (A) : (B))	
undefine	#undef <i>name</i>
quoted string in replace	#
Example. #define msg(A) printf("%s = %d", #A, (A))	
concatenate args and rescan	##
conditional execution	#if, #else, #elif, #endif
is <i>name</i> defined, not defined?	#ifdef, #ifndef
<i>name</i> defined?	defined(<i>name</i>)
line continuation char	\

Data Types/Declarations

character (1 byte)	char
integer	int
real number (single, double precision)	float, double
short (16 bit integer)	short
long (32 bit integer)	long
double long (64 bit integer)	long long
positive or negative	signed
non-negative modulo 2 ^m	unsigned
pointer to int, float,...	int*, float*,...
enumeration constant	enum <i>tag</i> { <i>name</i> ₁ = <i>value</i> ₁ ,...};
constant (read-only) value	<i>type</i> const <i>name</i> ;
declare external variable	extern
internal to source file	static
local persistent between calls	static
no value	void
structure	struct <i>tag</i> {...};
create new name for data type	typedef <i>type</i> <i>name</i> ;
size of an object (type is <i>size_t</i>)	sizeof <i>object</i>
size of a data type (type is <i>size_t</i>)	sizeof(<i>type</i>)

Initialization

initialize variable	<i>type</i> <i>name</i> = <i>value</i> ;
initialize array	<i>type</i> <i>name</i> []={ <i>value</i> ₁ ,...};
initialize char string	char <i>name</i> []="string";

Constants

suffix: long, unsigned, float	65536L, -1U, 3.0F
exponential form	4.2e1
prefix: octal, hexadecimal	0, 0x or 0X
Example. 031 is 25, 0x31 is 49 decimal	
character constant (char, octal, hex)	'a', '\ooo', '\xhh'
newline, cr, tab, backspace	\n, \r, \t, \b
special characters	\\, \?, \', \"
string constant (ends with '\0')	"abc...de"

Pointers, Arrays & Structures

declare pointer to <i>type</i>	<i>type</i> * <i>name</i> ;
declare function returning pointer to <i>type</i>	<i>type</i> *f();
declare pointer to function returning <i>type</i>	<i>type</i> (*pf)();
generic pointer type	void *
null pointer constant	NULL
object pointed to by <i>pointer</i>	* <i>pointer</i>
address of object <i>name</i>	& <i>name</i>
array	<i>name</i> [<i>dim</i>]
multi-dim array	<i>name</i> [<i>dim</i> ₁][<i>dim</i> ₂]...

Structures

struct <i>tag</i> {	structure template
<i>declarations</i>	declaration of members
};	

create structure	struct <i>tag</i> <i>name</i>
member of structure from template	<i>name</i> . <i>member</i>
member of pointed-to structure	<i>pointer</i> -> <i>member</i>
Example. (*p).x and p->x are the same	
single object, multiple possible types	union
bit field with <i>b</i> bits	unsigned <i>member</i> : <i>b</i> ;

Operators (grouped by precedence)

struct member operator	<i>name</i> . <i>member</i>
struct member through pointer	<i>pointer</i> -> <i>member</i>
increment, decrement	++, --
plus, minus, logical not, bitwise not	+, -, !, ~
indirection via pointer, address of object	* <i>pointer</i> , & <i>name</i>
cast expression to type	(<i>type</i>) <i>expr</i>
size of an object	sizeof
multiply, divide, modulus (remainder)	*, /, %
add, subtract	+, -
left, right shift [bit ops]	<<, >>
relational comparisons	>, >=, <, <=
equality comparisons	==, !=
and [bit op]	&
exclusive or [bit op]	^
or (inclusive) [bit op]	
logical and	&&
logical or	
conditional expression	<i>expr</i> ₁ ? <i>expr</i> ₂ : <i>expr</i> ₃
assignment operators	+=, -=, *=, ...
expression evaluation separator	,

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

statement terminator	;
block delimiters	{ }
exit from switch, while, do, for	break;
next iteration of while, do, for	continue;
go to	goto <i>label</i> ;
label	<i>label</i> : <i>statement</i>
return value from function	return <i>expr</i>

Flow Constructions

if statement	if (<i>expr</i> ₁) <i>statement</i> ₁ else if (<i>expr</i> ₂) <i>statement</i> ₂ else <i>statement</i> ₃
while statement	while (<i>expr</i>) <i>statement</i>
for statement	for (<i>expr</i> ₁ ; <i>expr</i> ₂ ; <i>expr</i> ₃) <i>statement</i>
do statement	do <i>statement</i> while(<i>expr</i>);
switch statement	switch (<i>expr</i>) { case <i>const</i> ₁ : <i>statement</i> ₁ break; case <i>const</i> ₂ : <i>statement</i> ₂ break; default: <i>statement</i> }

ANSI Standard Libraries

<assert.h>	<ctype.h>	<errno.h>	<float.h>	<limits.h>
<locale.h>	<math.h>	<setjmp.h>	<signal.h>	<stdarg.h>
<stddef.h>	<stdio.h>	<stdlib.h>	<string.h>	<time.h>

Character Class Tests <ctype.h>

alphanumeric?	isalnum(c)
alphabetic?	isalpha(c)
control character?	isctrl(c)
decimal digit?	isdigit(c)
printing character (not incl space)?	isgraph(c)
lower case letter?	islower(c)
printing character (incl space)?	isprint(c)
printing char except space, letter, digit?	ispunct(c)
space, formfeed, newline, cr, tab, vtab?	isspace(c)
upper case letter?	isupper(c)
hexadecimal digit?	isxdigit(c)
convert to lower case	tolower(c)
convert to upper case	toupper(c)

String Operations <string.h>

s is a string; cs, ct are constant strings

length of s	strlen(s)
copy ct to s	strcpy(s,ct)
concatenate ct after s	strcat(s,ct)
compare cs to ct	strcmp(cs,ct)
only first n chars	strncmp(cs,ct,n)
pointer to first c in cs	strchr(cs,c)
pointer to last c in cs	strrchr(cs,c)
copy n chars from ct to s	memcpy(s,ct,n)
copy n chars from ct to s (may overlap)	memmove(s,ct,n)
compare n chars of cs with ct	memcmp(cs,ct,n)
pointer to first c in first n chars of cs	memchr(cs,c,n)
put c into first n chars of s	memset(s,c,n)

C Reference Card (ANSI)

Input/Output <stdio.h>

Standard I/O

standard input stream	<code>stdin</code>
standard output stream	<code>stdout</code>
standard error stream	<code>stderr</code>
end of file (type is <code>int</code>)	<code>EOF</code>
get a character	<code>getchar()</code>
print a character	<code>putchar(<i>chr</i>)</code>
print formatted data	<code>printf("format",<i>arg</i>₁,...)</code>
print to string <i>s</i>	<code>sprintf(<i>s</i>, "format",<i>arg</i>₁,...)</code>
read formatted data	<code>scanf("format",&<i>name</i>₁,...)</code>
read from string <i>s</i>	<code>sscanf(<i>s</i>, "format",&<i>name</i>₁,...)</code>
print string <i>s</i>	<code>puts(<i>s</i>)</code>

File I/O

declare file pointer	<code>FILE *<i>fp</i>;</code>
pointer to named file	<code>fopen("name", "mode")</code> modes: <i>r</i> (read), <i>w</i> (write), <i>a</i> (append), <i>b</i> (binary)
get a character	<code>getc(<i>fp</i>)</code>
write a character	<code>putc(<i>chr</i>, <i>fp</i>)</code>
write to file	<code>fprintf(<i>fp</i>, "format",<i>arg</i>₁,...)</code>
read from file	<code>fscanf(<i>fp</i>, "format",<i>arg</i>₁,...)</code>
read and store <i>n</i> elts to * <i>ptr</i>	<code>fread(*<i>ptr</i>,<i>eltsize</i>,<i>n</i>,<i>fp</i>)</code>
write <i>n</i> elts from * <i>ptr</i> to file	<code>fwrite(*<i>ptr</i>,<i>eltsize</i>,<i>n</i>,<i>fp</i>)</code>
close file	<code>fclose(<i>fp</i>)</code>
non-zero if error	<code>ferror(<i>fp</i>)</code>
non-zero if already reached EOF	<code>feof(<i>fp</i>)</code>
read line to string <i>s</i> (< <code>max</code> chars)	<code>fgets(<i>s</i>,<i>max</i>,<i>fp</i>)</code>
write string <i>s</i>	<code>fputs(<i>s</i>,<i>fp</i>)</code>

Codes for Formatted I/O: "%-+ 0w.pmc"

-	left justify
+	print with sign
<i>space</i>	print space if no sign
0	pad with leading zeros
<i>w</i>	min field width
<i>p</i>	precision
<i>m</i>	conversion character:
	<i>h</i> short, <i>l</i> long, <i>L</i> long double
<i>c</i>	conversion character:
<i>d,i</i>	integer <i>u</i> unsigned
<i>c</i>	single char <i>s</i> char string
<i>f</i>	double (printf) <i>e,E</i> exponential
<i>f</i>	float (scanf) <i>lf</i> double (scanf)
<i>o</i>	octal <i>x,X</i> hexadecimal
<i>p</i>	pointer <i>n</i> number of chars written
<i>G,g</i>	same as <i>f</i> or <i>e,E</i> depending on exponent

Variable Argument Lists <stdarg.h>

declaration of pointer to arguments	<code>va_list <i>ap</i>;</code>
initialization of argument pointer	<code>va_start(<i>ap</i>,<i>lastarg</i>);</code> <i>lastarg</i> is last named parameter of the function
access next unnamed arg, update pointer	<code>va_arg(<i>ap</i>,<i>type</i>)</code>
call before exiting function	<code>va_end(<i>ap</i>);</code>

Standard Utility Functions <stdlib.h>

absolute value of <code>int</code> <i>n</i>	<code>abs(<i>n</i>)</code>
absolute value of <code>long</code> <i>n</i>	<code>labs(<i>n</i>)</code>
quotient and remainder of ints <i>n,d</i>	<code>div(<i>n</i>,<i>d</i>)</code> returns structure with <code>div_t.quot</code> and <code>div_t.rem</code>
quotient and remainder of longs <i>n,d</i>	<code>ldiv(<i>n</i>,<i>d</i>)</code> returns structure with <code>ldiv_t.quot</code> and <code>ldiv_t.rem</code>
pseudo-random integer [0,RAND_MAX]	<code>rand()</code>
set random seed to <i>n</i>	<code>srand(<i>n</i>)</code>
terminate program execution	<code>exit(<i>status</i>)</code>
pass string <i>s</i> to system for execution	<code>system(<i>s</i>)</code>
Conversions	
convert string <i>s</i> to double	<code>atof(<i>s</i>)</code>
convert string <i>s</i> to integer	<code>atoi(<i>s</i>)</code>
convert string <i>s</i> to long	<code>atol(<i>s</i>)</code>
convert prefix of <i>s</i> to double	<code>strtod(<i>s</i>,&<i>endp</i>)</code>
convert prefix of <i>s</i> (base <i>b</i>) to long	<code>strtoul(<i>s</i>,&<i>endp</i>,<i>b</i>)</code>
same, but unsigned long	<code>strtoul(<i>s</i>,&<i>endp</i>,<i>b</i>)</code>

Storage Allocation

allocate storage	<code>malloc(<i>size</i>), calloc(<i>nobj</i>,<i>size</i>)</code>
change size of storage	<code>newptr = realloc(<i>ptr</i>,<i>size</i>);</code>
deallocate storage	<code>free(<i>ptr</i>);</code>

Array Functions

search array for key	<code>bsearch(<i>key</i>,<i>array</i>,<i>n</i>,<i>size</i>,<i>cmpf</i>)</code>
sort array ascending order	<code>qsort(<i>array</i>,<i>n</i>,<i>size</i>,<i>cmpf</i>)</code>

Time and Date Functions <time.h>

processor time used by program	<code>clock()</code>
<i>Example.</i> <code>clock()/CLOCKS_PER_SEC</code> is time in seconds	
current calendar time	<code>time()</code>
<i>time</i> ₂ - <i>time</i> ₁ in seconds (double)	<code>difftime(<i>time</i>₂,<i>time</i>₁)</code>
arithmetic types representing times	<code>clock_t</code> , <code>time_t</code>
structure type for calendar time comps	<code>struct tm</code>
<code>tm_sec</code>	seconds after minute
<code>tm_min</code>	minutes after hour
<code>tm_hour</code>	hours since midnight
<code>tm_mday</code>	day of month
<code>tm_mon</code>	months since January
<code>tm_year</code>	years since 1900
<code>tm_wday</code>	days since Sunday
<code>tm_yday</code>	days since January 1
<code>tm_isdst</code>	Daylight Savings Time flag

convert local time to calendar time	<code>mktime(<i>tp</i>)</code>
convert time in <i>tp</i> to string	<code>asctime(<i>tp</i>)</code>
convert calendar time in <i>tp</i> to local time	<code>ctime(<i>tp</i>)</code>
convert calendar time to GMT	<code>gmtime(<i>tp</i>)</code>
convert calendar time to local time	<code>localtime(<i>tp</i>)</code>
format date and time info	<code>strftime(<i>s</i>,<i>smax</i>, "format", <i>tp</i>)</code> <i>tp</i> is a pointer to a structure of type <code>tm</code>

Mathematical Functions <math.h>

Arguments and returned values are double

trig functions	<code>sin(x), cos(x), tan(x)</code>
inverse trig functions	<code>asin(x), acos(x), atan(x)</code>
<code>arctan(<i>y/x</i>)</code>	<code>atan2(<i>y</i>,<i>x</i>)</code>
hyperbolic trig functions	<code>sinh(x), cosh(x), tanh(x)</code>
exponentials & logs	<code>exp(x), log(x), log10(x)</code>
exponentials & logs (2 power)	<code>ldexp(x,<i>n</i>), frexp(x,&<i>e</i>)</code>
division & remainder	<code>modf(x,<i>ip</i>), fmod(x,<i>y</i>)</code>
powers	<code>pow(x,<i>y</i>), sqrt(x)</code>
rounding	<code>ceil(x), floor(x), fabs(x)</code>

Integer Type Limits <limits.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system, followed by minimum required values (if significantly different).

<code>CHAR_BIT</code>	bits in char	(8)
<code>CHAR_MAX</code>	max value of char	(<code>SCHAR_MAX</code> or <code>UCHAR_MAX</code>)
<code>CHAR_MIN</code>	min value of char	(<code>SCHAR_MIN</code> or 0)
<code>SCHAR_MAX</code>	max signed char	(+127)
<code>SCHAR_MIN</code>	min signed char	(-128)
<code>SHRT_MAX</code>	max value of short	(+32,767)
<code>SHRT_MIN</code>	min value of short	(-32,768)
<code>INT_MAX</code>	max value of int	(+2,147,483,647) (+32,767)
<code>INT_MIN</code>	min value of int	(-2,147,483,648) (-32,767)
<code>LONG_MAX</code>	max value of long	(+2,147,483,647)
<code>LONG_MIN</code>	min value of long	(-2,147,483,648)
<code>UCHAR_MAX</code>	max unsigned char	(255)
<code>USHRT_MAX</code>	max unsigned short	(65,535)
<code>UINT_MAX</code>	max unsigned int	(4,294,967,295) (65,535)
<code>ULONG_MAX</code>	max unsigned long	(4,294,967,295)

Float Type Limits <float.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.

<code>FLT_RADIX</code>	radix of exponent rep	(2)
<code>FLT_ROUNDS</code>	floating point rounding mode	
<code>FLT_DIG</code>	decimal digits of precision	(6)
<code>FLT_EPSILON</code>	smallest <i>x</i> so $1.0f + x \neq 1.0f$	($1.1E - 7$)
<code>FLT_MANT_DIG</code>	number of digits in mantissa	
<code>FLT_MAX</code>	maximum float number	(3.4E38)
<code>FLT_MAX_EXP</code>	maximum exponent	
<code>FLT_MIN</code>	minimum float number	($1.2E - 38$)
<code>FLT_MIN_EXP</code>	minimum exponent	
<code>DBL_DIG</code>	decimal digits of precision	(15)
<code>DBL_EPSILON</code>	smallest <i>x</i> so $1.0 + x \neq 1.0$	($2.2E - 16$)
<code>DBL_MANT_DIG</code>	number of digits in mantissa	
<code>DBL_MAX</code>	max double number	(1.8E308)
<code>DBL_MAX_EXP</code>	maximum exponent	
<code>DBL_MIN</code>	min double number	($2.2E - 308$)
<code>DBL_MIN_EXP</code>	minimum exponent	

January 2007 v2.2. Copyright © 2007 Joseph H. Silverman

Permission is granted to make and distribute copies of this card provided the copyright notice and this permission notice are preserved on all copies.

Send comments and corrections to J.H. Silverman, Math. Dept., Brown Univ., Providence, RI 02912 USA. (jhs@math.brown.edu)