PicoWeb Pcode

Introduction

A *pcode* instruction interpreter was developed for the PicoWeb Server. The use of a custom pcode provides:

- program code simplification,
- the option to execute program code out of EEPROM (including external serial EEPROM), and
- in many cases a reduction in program code size as compared to native code.

Code simplification is achieved because the p-code makes no reference to native registers, and because the pcode "virtual machine" uses 16-bit wide data types for most operands. Pcode execution from EEPROM is possible because the pcode "instruction pointer" is 16 bits wide, more than is needed to address the program memory in the Atmel microcontroller. Therefore, the “extra” address bits can be used to indicate that the next pcode instruction should be fetched from EEPROM, and not from the microcontroller’s internal program memory.

PicoWeb Pcode Interpreter

Pcode Virtual Machine - The pcode virtual machine has no explicit registers, with the exception of a pcode *flags register* and a pcode *program counter*. Instead, most pcode instructions reference SRAM locations in the microcontroller. The same memory location labels used as part of normal AVR assembly language programming can be used as part of a pcode instruction instance. For example, the pcode instruction:

```
pincw buf
```

increments the two-byte value stored at SRAM label `buf`. By convention, the PicoWeb server’s standard pcode uses an n-byte scratch buffer located starting at SRAM location `buf` for general purpose working storage, a kind of pcode “register set”.

The pcode interpreter maintains a separate 5-deep pcode return address stack which is used store return address needed to implement the `pcall` and `pret` pcode instructions.

Refer to the section *PicoWeb Pcode Instruction Definitions* for a list of legal pcode instructions along with a description of their required operands. Pcode instructions can have a maximum of three operands. Most pcode built-in instructions work with 16-bit *words*. Like the AVR microcontroller, these 16-bit words are stored in memory in “little-endian” format (i.e., low byte is stored first in memory).

PicoWeb pcode can be freely intermixed with AVR assembly language *providing* that a `pbegin` pcode instruction us used to enter a pcode section and a `pend` pcode instruction is used to exit a pcode section. Failure to observe this rule will produce unpredictable results. (The pcode instruction `pbegin` is an AVR assembly macro for `pcall pcode` and the pcode instruction `pend` is a macro for `.dw 0`.)

User-Supplied Pcode Opcodes - Users can add their own pcode opcodes, providing certain conventions are followed. The following example shows “skeleton” code for a user-supplied pcode instruction called `mypcode` that takes three operands:

```
mypcode: pcode_routine 3
         ;
         ;   (user-supplied code goes here)
         ;
         ret   ; return to pcode interpreter
```

The first line of this sample routine tells the PicoWeb development system that `mypcode` is a new, legal pcode instruction which requires exactly three operands.

Note that when invoked by the pcode interpreter, the pcode instruction’s three 16-bit operands are supplied to the user-supplied assembly language routine in registers r10-r11, r12-r13 and r14-r15 respectively. These operands will

have been previously “de-referenced” by the pcode interpreter as indicated by the various operand word control bits (i.e., optional indirect addressing of words/bytes, byte swap, etc.).

A user-supplied pcode opcode assembly routine is free to use the processor registers r0-r1, r10-r17, x, y, and z. Any other processor registers need to be saved and restored before exiting the user’s routine. A user-supplied pcode opcode routine cannot depend upon any of the processor registers being preserved from one pcode opcode routine execution to the next. If a user-supplied pcode opcode routine needs to preserve state between opcode invocations, then the microcontroller SRAM must be used to save this state.

A user-supplied pcode routine can access to the pcode virtual machine’s *pcode flags register*. The format of the pcode flags register is identical to the AVR microcontroller’s flags register. A user-supplied pcode routine can copy the pcode flags register into the AVR flags register with the assembly language instruction “`rcall get_pcode_flags`”. The current state of the AVR flags register can be copied into the pcode flags register with the assembly language instruction “`rcall set_pcode_flags`”.

Pcode Instruction Memory Format – Pcode source instructions are pre-processed by the PicoWeb development environment and converted into a series of AVR assembler `.dw` pseudo-ops. Each pcode instruction generates a single 16-bit *pcode opcode word* followed by zero or more 16-bit *pcode operand words*. The bottom 12 bits of the opcode word are the address of the native execution routine. The uppermost bit of the opcode word is an "extended addressing" flag. If this bit is set, extended addressing is enabled, otherwise it is disabled. Extended addressing controls how the operand words are interpreted. When extended addressing is disabled, all operand words are considered "immediate" operands. When extended addressing is enabled, the uppermost three bits in each operand word specify which extended addressing mode(s) are to be applied, with the lower 13 bits of the operand treated as an address.

Pcode Opcode Word Format

Opcode Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Extended addressing enabled	1	0	0	0	A	A	A	A	A	A	A	A	A	A	A	A
Extended addressing disabled	0	0	0	0	A	A	A	A	A	A	A	A	A	A	A	A

AAAAAAAAAAAA = address of pcode routine in on-chip or external EEPROM memory

Pcode Operand Word Format

Operand Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Extended addressing enabled	0	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
	X	W	S	A	A	A	A	A	A	A	A	A	A	A	A	A
Extended addressing disabled	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I

IIIIIIIIIIIIIII = immediate data value
AAAAAAAAAAAA = address of pcode routine in on-chip or external EEPROM memory
W = 0: fetch 16-bit data word at address AAAAAAAAAAAA (indirect word addressing)
W = 1: fetch 8-bit data byte at address AAAAAAAAAAAA (indirect byte addressing)
S = 0: do nothing after fetching data
S = 1: swap high/low bytes after fetching data

A pcode opcode word of all zeros (`.dw 0`) will cause the pcode interpreter to exit and jump to the first AVR assembly language instruction after the zero word. (The pcode opcode `pend` is defined to have all 16 opcode word bits set to zero.)

PicoWeb Pcode Example

The following example is a pcode routine called `decconv` which converts a passed 16-bit signed integer value into ASCII digits, then outputs that text (to the serial port or to the Ethernet) using the pcode I/O opcode `pputcb`. This "pure pcode" routine uses 11 bytes of SRAM starting at the label `buf` to perform its work. It can be called from pcode as in the following pcode example, which will output the string "-12345":

```
pmovwi  buf,-12345
pcall   decconv
```

The routine `decconv` can be placed in the AVR microprocessor's on-chip program memory, or it can be placed in the PicoWeb server's external serial EEPROM memory. Execution speed is much slower from the external SEEPROM memory because in that case the opcode and operand words must be fetched serially as they are executed (i.e., one bit at a time at the rate of ~400 Kbits/sec over the I^2C bus).

```
;
; decconv – integer to decimal converter
;
; inputs:
;   buf = 16-bit signed integer to convert
;
; outputs:
;   ASCII value, printed with pputc
;
; method:
;   repeatedly divide input by 10 until quotient is 0, using the remainder
;   to make ASCII digits.  The ASCII digits are buffered, then output at
;   end in reverse order.  Negative inputs get a leading '-' sign.
;
; caveats:
;   MUST be called with a pcall!!
;   Destroys SRAM locations buf through buf+10
;
#define DEC_VAL buf                  ; word to convert (destroyed!)
#define DEC_REM buf+2                ; remainder from divide op
#define DIG_PTR buf+4                ; text buffer pointer
#define DIG_BUF buf+6                ; text buffer (5 bytes max.)

decconv:
    pmovwi DIG_PTR,DIG_BUF           ; initialize pointer to DIG_BUF
    pbitwi DEC_VAL,0x8000            ; what's the sign of input?
    pjumpeq decconv1                 ; positive...start conversion
    pnegw DEC_VAL                    ; negative...negate input word
    pputc '-'                        ; output leading minus sign
decconv1:
    pdiv DEC_VAL,[DEC_VAL],10        ; do next digit (divide by 10)
    paddwi DEC_REM,'0'               ; convert digit for display
    pmovbi [DIG_PTR],[byte DEC_REM]  ; save ASCII digit
    pincw DIG_PTR                    ; bump past digit just stored
    psubwi DEC_VAL,0                 ; test remaining value
    pjumpne decconv1                 ; more to convert if non-zero
decconv2:
    pdecw DIG_PTR                    ; point to next digit to output
    pputcb [byte DIG_PTR]            ; output ASCII digit
    pcmpwi DIG_PTR,DIG_BUF           ; are we back to the beginning?
    pjumpne decconv2                 ; nope...do another digit
    pret                             ; yes...return to caller
```

Preprocessor `#define` statements are used in the example to make the source code more readable. These `#define` statements call out SRAM locations in the PicoWeb server's general purpose "scratch" buffer `buf`.

Note the use of square brackets ([ ]) on three of the sample pcode instruction lines indicating “indirect addressing”. For example, the `pmovbi` pcode opcode (move byte) expects a destination memory address as its first operand and an immediate value as its second operand. In the example, the destination address is specified as `[DIG_PTR]`, an *indirect addressing* form, which tells the pcode interpreter to instead use `DIG_PTR` as a pointer to the “real” target byte address. The routine increments DIG_PTR as it converts digits, allowing the ASCII digits to be stored sequentially starting at memory location DIG_BUF.

On the same sample pcode `pmovbi` instruction line, the second operand normally supplies an *immediate* source byte. However, in the example, this operand is specified as `[byte DEC_REM]`, a more complex *indirect addressing* mode, which tells the pcode interpreter to fetch the byte stored at memory location `DEC_REM`, instead of simply taking the immediate value supplied as the second pcode operand. If the second operand was specified as simply “`DEC_REM`”, then the pcode instruction would (incorrectly) repeatedly store the low byte of the literal address `DEC_REM`!

PicoWeb Pcode Instruction Definitions (v1.36)

All pcode instructions take one of the following forms:

opcode	
opcode	*p1*
opcode	*p1,p2*
opcode	*p1,p2,p3*

where:

opcode	pcode opcode listed in one of the PicoWeb pcode instruction tables
p1-p3	pcode *operand* (p1, p2, p3) shown in the PicoWeb pcode instruction tables

All pcode operands (*p1, p2, p3*) may be expressed in one of seven different forms, one *normal form*, three different *indirect forms*, and three different *string literal* forms, as follows:

p	effective operand is 16-bit value *p* (normal form)
[*p*]	effective operand is 16-bit word stored beginning at SRAM address *p*
[byte *p*]	effective operand is 8-bit byte stored at SRAM address *p*
[swap *p*]	effective operand is 16-bit word at SRAM address *p* with high/low bytes swapped
"..."	effective operand is address of character string ("...") in the current code segment (.cseg or .eseg)
cseg "..."	effective operand is address of character string ("...") in program memory (.cseg)
eseg "..."	effective operand is address of character string ("...") in external SEEPROM memory (.eseg)

Warning: If *any* of the *indirect* operand forms are used in a given pcode instruction instance, then the size of any (and all) immediate (i.e., *imm* operands) operands used in that pcode instruction instance is reduced from 16 bits to 14 bits.

Notes regarding the PicoWeb pcode tables which follow:

- *a* indicates an SRAM address
- **a* indicates the 16-bit word starting at SRAM byte address *a*
- *a[n]* indicates the contents of the SRAM byte at address *(a + n)*
- *eeprom[e]* indicates the contents of the on-chip EEPROM byte at address *e*
- *seeprom[e]* indicates the contents of the external serial EEPROM byte at address *e*
- *imm* is an immediate 16-bit word value
- *ppc* signifies the pcode program counter
- *s* signifies the start address of a zero-terminated string in on-chip flash memory or external SEEPROM memory
- addr signifies the address of a pcode opcode (in on-chip flash or external serial EEPROM memory)
- The pcode flags (Z, C, N) shown in the "**Flags**" column are separate from the AVR processor flags
- Z is the pcode zero flag
- C is the pcode carry flag
- N is the pcode negative flag (i.e., sign bit)
- *low(x)* indicates the low 8 bits of a 16-bit word value *x*
- *swap(a)* indicates an exchange of the low and high bytes of a 16-bit word *a*

PicoWeb Pcode General Purpose Instructions

Opcode	p1	p2	p3	Description	Operation	Flags
paddwi	a	imm		Add immediate to word	∗a ← ∗a + imm	Z,C,N
pandwi	a	imm		Logical AND word immediate	∗a ← ∗a & imm	Z,N
pbegin				Begin executing pcode	rcall pcode	
pbitwi	a	imm		Bit test word immediate.	∗a & imm	Z
pcall	addr			Call pcode routine	push *ppc* onto pcode return stack; *ppc* ¬ *addr*	
pclrw	a			Clear word	∗a ← 0	
pcmpbi	a	imm		Compare byte immediate	a[0] - low(imm)	Z,C,N
pcmpn	a	b	n	Compare two buffers in SRAM	for (i=0; i<n; ++i) a[i] - b[i];	Z
pcmpwi	a	b		Compare word immediate.	∗a - imm	Z,C,N
pcomw	a			Ones complement word.	∗a ← 0xFFFF - ∗a	Z,C,N
pdecw	a			Decrement word	∗a ← ∗a - 1	Z,C,N
pdiv	a	b	c	Unsigned 16-bit divide, 32-bit result	∗a = ∗b / ∗c; ∗(a+2) = ∗b % ∗c;	
pee2s	a	e	n	Copy bytes from EEPROM to SRAM	for (i=0; i<n; ++i) a[i] ← eeprom[e + n];	
pend				Stop executing pcode	.dw 0	
pincw	a			Increment word	∗a ← ∗a + 1	Z,C,N
pjump	addr			Unconditional jump within pcode	ppc ← addr	
pjumpeq	addr			Jump within pcode if equal	if (Z == 1) ppc ← addr	
pjumplo	addr			Jump within pcode if lower	if (N == 1) ppc ← addr	
pjumpne	addr			Jump within pcode if not equal	if (Z == 0) ppc ← addr	
pmemcpy	a	b	n	SRAM memory copy	for (i=0; i<n; ++i) a[i] ← b[i];	
pmovb	a	b		Move byte	a[0] ← b[0]	
pmovbi	a	imm		Move byte immediate	a[0] ← low(imm)	
pmovwi	a	imm		Move word immediate	*a ← imm	
pmul	a	b	c	Unsigned 16-bit multiply, 32-bit result	*a ← (*b × *c) & 0xFFFF; *(a+2) ← (*b × ∗c) >> 16	
pnegw	a			Two's complement word	∗a ← 0 - ∗a	Z,C,N
ppopw	a			Pop word from stack	∗a ← pop top two bytes from AVR stack	
ppushwi	imm			Push immediate word onto stack	push low(*imm*), high(*imm*) to top of AVR stack	
pret				Return from pcode routine	ppc ← pop pcode return stack	
ps2ee	e	a	n	Copy bytes from SRAM to EEPROM	for (i=0; i<n; ++i) eeprom[e + n] ← a[i];	
ps2see	e	a	n	Copy bytes from SRAM to ext. SEEPROM	for (i=0; i<n; ++i) seeprom[e + n] ← a[i];	
psee2s	a	e	n	Copy bytes from ext. SEEPROM to SRAM	for (i=0; i<n; ++i) a[i] ← seeprom[e + n];	
pshnw	a	n		Logical shift 16-bit word *n* bits	if (n > 0) ∗a ← (∗a << low(n)) else ∗a ← (∗a >> low(n));	Z,C,N
psubwi	a	imm		Subtract word immediate	∗a ← ∗a - imm	Z,C,N
pwdr				Watchdog timer reset	wdr	
pwreebi	e	imm		Write byte to EEPROM	eeprom(e) ← low(imm)	
pwrseebi	e	imm		Write byte to ext. SEEPROM	seeprom(e) ← low(imm)	
pxorwi	a	imm		Exclusive OR word immediate	∗a ← ∗a ^ imm, set flags with low(∗a)	Z,N

Notes:

- A pcode transfer of control instruction (e.g., pcall, pjump, etc.) *must have* as its target address another pcode instruction. Transfering control from pcode directly to AVR assembly language will produce unpredictable results!
- Pcode *must be* entered from AVR assembly language using a *pbegin* instruction. Return to AVR assembly language *must be* done using a *pend* instruction. Unpredictable results can be expected if these rules are not followed!

PicoWeb Pcode Input/Output Instructions

Opcode	p1	p2	p3	Description	Operation	Flags
pcrlf				Print CR,LF	printf("\r\n")	
phexbi	a			Print hex byte immediate	printf("%02x", (a & 0xFF))	
pprint	s			Print string	printf("%s", a)	
pprintb	s	a	n	Print bytes in hex with string	if (s != 0) printf("%s", s); for (i=0; i<n; ++i) printf("%02x", a[i]);	
pprintv	s	imm		Print string with 16-bit hex value	if (s != 0) printf("%s", s) else putchar(' '); printf("%04x", imm);	
pputc	imm			Print character immediate	putchar(imm & 0xFF);	
pputcb	a			Print character	putchar(a[0] & 0xFF);	
pser_getc	a			Check/get serial port character	if (character ready) a ← getchar(), Z = 0 else Z = 1;	Z
pser_mode	bin			Flush and set serial port mode	Flush any buffered characters, then if (*bin* = 0) set normal mode; else set pass-all (binary) mode	
pser_putc	imm			Write immediate byte to serial port	Wait for a "transmit done" indication on serial port, then write *low(imm)* to UART	
pspace				Print a space	putchar(' ')	

Notes:

- The operation *putchar(ch)* behaves as follows: If (putc_b == 0), the byte *ch* is written to serial port,. If (putc_b ≠ 0), the byte *ch* is sent to transmit stream or stored in holding buffer pending flush of transmit buffer.
- The operation *printf()* behaves identically as the previously described *putchar()* operation

PicoWeb Pcode Network Instructions

Opcode	p1	p2	p3	Description	Operation	Flags
paddn	a	b	n	Add n-byte integers (network byte ordering)	CF = 0; for (i=low(n)/2-1; i≠0; i-=1) *(a+2*i) ← swap(swap(*(a+2*i)) + swap(*(b+2*i)) + CF);	
pf2x	off	func	n	Output bytes to Ethernet transmit buffer starting at byte *off* with checksum	for (i=0; i<n; ++i) Xmit[off + i] ← func(); where *func* returns 16-bit word in X register. A 16-bit network checksum accumulated into "chkacc".	
pi2x	off	imm		Output immediate 16-bit word to Ethernet transmit buffer at byte *off*	Xmit[off] ← imm	
pprinta	s	eadd	n	Print bytes from Ethernet receive buffer with label	if (s != 0) printf("%s=", s); for (i=0; i<n; ++i) putchar(Recv[eadd + i]);	
pprintr	s	off	n	Print words in hex from current receive buffer with label	if (s != 0) printf("%s=", s); for (i=0; i<2*n; ++i) printf("%02x", Recv[off + i]);	
pprintt	s	off	n	Print words in hex from current transmit buffer with label	if (s != 0) printf("%s=", s); for (i=0; i<2*n; ++i) printf("%02x", Xmit[off + i]);	
pr2s	a	off	n	Move bytes from current Ethernet receive buffer to memory	for (i=0; i<n; ++i) a[i] ← Recv[off + i];	
pr2x	off1	off2	n	Move bytes from current Ethernet receive buffer to Ethernet transmit buffer.	for (i=0; i<n; ++i) Xmit[off1 + i] ← Recv[off2 + i];	
ps2x	off	a	n	Move bytes from memory to Ethernet transmit buffer	for (i=0; i<n; ++i) Xmit[off + i] ← a[i];	
px2s	a	off	n	Move bytes from Ethernet transmit buffer to memory	for (i=0; i<n; ++i) a[i] ← Xmit[off + i];	
pz2x	off	n		Write *n* bytes of zero to transmit buffer	for (i=0; i<n; ++i) Xmit[off + i] ← 0;	

Notes:

- Xmit[0] is the first byte of the Ethernet transmit buffer and Recv[0] is the first byte of the Ethernet receive buffer. Both point to the first byte of the Ethernet MAC address in the respective buffer (i.e., any NIC-specific header bytes are skipped).
- The operation *printf()* behaves identically as the previously described *putchar()* operation

10/23/99 11:28 AM