

Micromint Modules



Microcomputer/controller with embedded BASIC interpreter

FEATURES

- Small size—complete computer/controller with I/O in less than 1.5 cubic inches (1.5" × 2.1" × 0.5")
- Low power—only 100 mW typical
- Dual powered—operates on +5 V or 8–16 V at 15 mA (typical)
- Communications through RS-232A, RS-422, or RS-485 serial port up to 19.2 kbps; internal on-chip level shifters
- Full floating-point BASIC for easy programming
- Two firmware PWM outputs—2 Hz–3 kHz, 5–95% duty cycle
- Hardware PWM output—up to 1 MHz, depending on duty cycle
- Frequency measurement—15 Hz–15 kHz
- I²C bus
- 32-KB SRAM for "enter and execute" program testing
- 32-KB EEPROM nonvolatile storage for autostart applications
- Hardware real-time clock/calendar
- 40-pin DIP-style enclosed packaging with rugged square pins
- Optional 2-channel, 12-bit ADC, 7000 samples/second BASIC and Assembly, 250 samples/second straight BASIC
- 11.059-MHz system clock
- 2 interrupts and 3 timers
- Parallel I/O—(Processor) 12 bits of bit-programmable TTL-level and 16 bits of bit-programmable high current drive I/O lines; 25mA sink per pin with 20mA source per pin. Port A & B can only sink or source up to a combined total of 200mA.
- Regulated 5-V output powers external circuitry

DESCRIPTION

The **DOMINO-2** micro controller is a rugged, miniature controller with a fast, control-oriented, processor-masked BASIC interpreter. **DOMINO-2** programs can be entirely BASIC or a mixture of BASIC and assembly language routines with a BASIC CALL instruction.

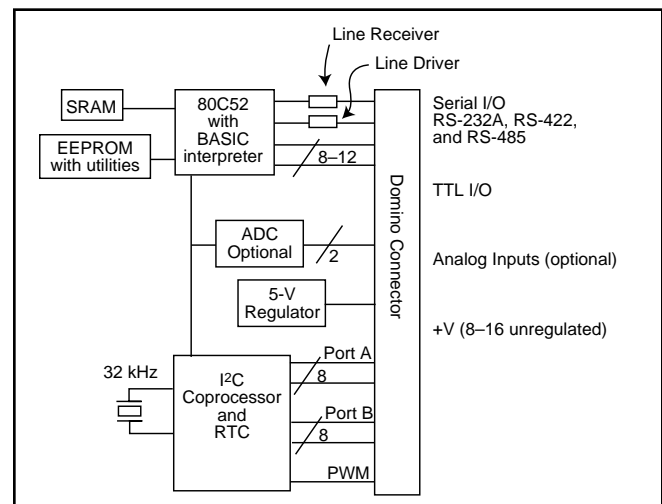
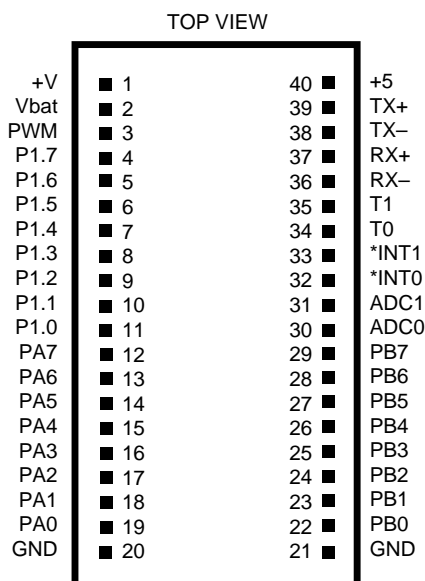
DOMINO-2 is designed to be a 100% stand-alone, low-power, embedded controller, which only requires a user to apply power to function. Power can be either +5 V only or +8-16 V to the internal regulator. When using the regulator input, a 5-V output is available to external circuitry.

DOMINO-2 is both RS-232A and RS-485 compatible without extra components. Based on a CMOS 80C52 processor, **DOMINO-2** provides a ROM-resident BASIC interpreter, 32-KB of static RAM, and 32 KB of nonvolatile EEPROM.

DOMINO-2 has 12 bi-directional bit-programmable parallel I/O lines (3 are shared with the ADC) plus 16 bits of bi-directional, bit-programmable high-current I/O lines provided by a built-in I²C parallel expander. These bits can source/sink more than 20 mA each (limited by total package dissipation). The coprocessor also offers a hardware PWM output and a real-time clock/calendar. Analog input is via a 2-channel sample-and-hold 12-bit ADC. It is capable of sampling at 7000 samples/sec BASIC & Assembly, 250 samples/sec BASIC.

Additional firmware enables program calls to directly read frequency and period, set PWM pulse width and duty cycle, communicate with I²C bus peripherals, and save programs to EEPROM that can be auto started.

DOMINO-2 combines the ROM-resident BASIC with a selection of firmware program calls to directly read frequency and period, set PWM pulse width and duty cycle, communicate with I²C bus peripherals, and save programs to EEPROM that can be auto started.



ABSOLUTE MAXIMUM RATINGS

Operating temperature:		Voltage on +5V (Pin 40)	0 to +5.5 V
Commercial	0°C to +70°C	referenced to Vss	
Industrial	–40°C to +85°C	with Pin 1 open	
Storage temperature	–50°C to +125°C	Voltage on Vbat (Pin 2)	0 to +5.0 V
Voltage on V+ (Pin 1)	0 to +16 V		
referenced to Vss			

Industrial temperature version is available; minimum quantities apply.

PIN DESCRIPTIONS

Domino-2 is a 40-pin package (2.25" × 1.4" × 0.5") with 0.1" pin and 1.2" row spacing. Some pins have multiple functions depending on system configuration.

Pin	Signal	Description	Pin	Signal	Description
1	V+	Domino-2 power-supply input. V+ is nominally 8–16 V. If pin 1 is open, Domino-2 can be +5-V powered directly to pin 40.	12	PA7	I2C Expansion I/O Port A bit 7; high-current I/O pin
2	Vbat	3V battery backup input for real-time clock See ERRATA on Page 27.	13	PA6	I2C Expansion I/O Port A bit 6; high-current I/O pin
3	PWM	Hardware PWM output generated by coprocessor	14	PA5	I2C Expansion I/O Port A bit 5; high-current I/O pin
4	P1.7	TTL I/O bit 7, available directly through BASIC; optionally used as ADC CS input and as I2C clock	15	PA6	I2C Expansion I/O Port A bit 4; high-current I/O pin
5	P1.6	TTL I/O bit 6, available directly through BASIC; optionally used as DATA I/O for ADC and I2C	16	PA3	I2C Expansion I/O Port A bit 3; high-current I/O pin
6	P1.5	TTL I/O bit 5, available directly through BASIC; optionally used as ADC CLK	17	PA2	I2C Expansion I/O Port A bit 2; high-current I/O pin.
7	P1.4	TTL I/O bit 4, available directly through BASIC	18	PA1	I2C Expansion I/O Port A bit 1; high-current I/O pin.
8	P1.3	TTL I/O bit 3, available directly through BASIC	19	PA0	I2C Expansion I/O Port A bit 0; high-current I/O pin.
9	P1.2	TTL I/O bit 2, available directly through BASIC	20	GND	Single point analog and digital ground.
10	P1.1	TTL I/O bit 1, available directly through BASIC	21	GND	Single point analog and digital ground.
11	P1.0	TTL I/O bit 0, available directly through BASIC	22	PB0	I2C Expansion I/O Port B bit 0; high-current I/O pin.
			23	PB1	I2C Expansion I/O Port B bit 1; high-current I/O pin.
			24	PB2	I2C Expansion I/O Port B bit 2; high-current I/O pin.

Pin Signal Description

25	PB3	I ² C Expansion I/O Port B bit 3; high-current I/O pin.
26	PB4	I ² C Expansion I/O Port B bit 4; high-current I/O pin.
27	PB5	I ² C Expansion I/O Port B bit 5; high-current I/O pin.
28	PB6	I ² C Expansion I/O Port B bit 6; high-current I/O pin.
29	PB7	I ² C Expansion I/O Port B bit 7; high-current I/O pin.
30	ADC0	12-bit ADC channel 0 input, input range 0–5 V.
31	ADC1	12-bit ADC channel 1 input, input range 0–5 V.
32	INT0	TTL Interrupt 0 input and general I/O bit (available through assembly language).
33	INT1	TTL Interrupt 1 input and general I/O bit (available through assembly language or BASIC).

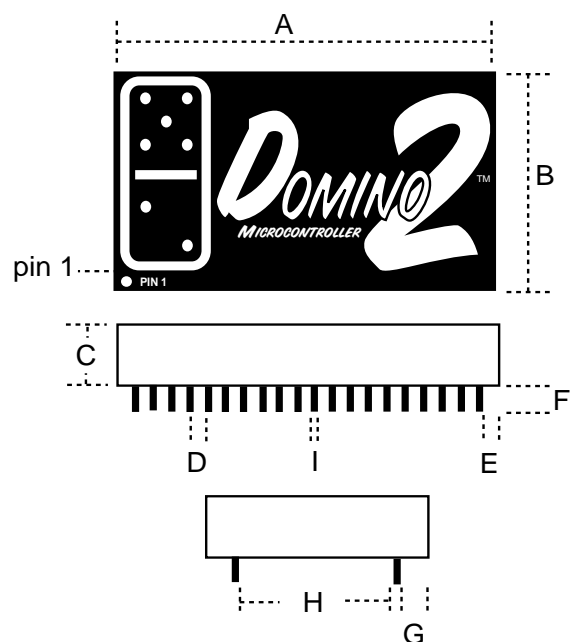
Pin Signal Description

34	T0	Serial transmitter disable control, TTL timer/counter input and general purpose I/O bit (available through assembly language).
35	T1	TTL timer/counter input and general purpose I/O bit (available through assembly language).
36	RX–	RS-422/-485/-232A inverted serial (receive pair/recxmit pair/receive)
37	RX+	RS-422/-485/-232A noninverted serial (receive pair/rec-xmit pair)
38	TX–	RS-422/-485/-232A inverted serial (transmit pair/rec-xmit pair/transmit)
39	TX+	RS-422/-485/-232A noninverted serial (transmit pair/rec-xmit pair)
40	+5V	This is the internal ADC voltage reference (nominally 5.0 V) This output may be used to power minimal external circuitry or sensors. DOMINO-2 may be powered on +5-V only through this pin, provided Pin 1 is left unconnected.

MECHANICAL AND ENVIRONMENTAL CHARACTERISTICS

Length	2.25 inches
Width	1.375 inches
Height	0.52 inches
Weight	25 grams
Operating temperature	0 to +70°C (optional –40 to +85 °C)
Humidity	0 to 100% (noncondensing)

DIM	Inches		Millimeters	
	min	max	min	max
A	2.240	2.260	56.896	57.404
B	1.365	1.385	34.671	35.179
C	0.510	0.530	12.954	13.462
D	0.095	0.105	2.413	2.667
E	0.153	0.193	3.886	4.902
F	0.220	0.280	5.588	7.112
G	0.170	0.200	4.318	5.080
H	0.990	1.100	25.146	27.940
I	0.023	0.027	0.508	0.686



DC ELECTRICAL CHARACTERISTICS

Operating temperature
Operating voltage

Ta = 0°C to +70°C
Vcc = 4.75 V to 5.25 V
Vss = 0.0 V

Characteristic	Minimum	Typical	Maximum	Units	Condition
Supply Voltage (Vcc to Pin 40)) (+V to Pin 1)	4.75 7	5.00 9	5.25 16	V V	
Supply Current (Icc) (RS-422/485 50-Ω termination disabled)		35-50mA			mAtypical
Input Low Voltage (Vil)	-0.5		0.9	V	
Input High Voltage (Vih)	1.9		5.5	V	
Output Low Voltage (Vol)		0.45		V	Iol=1.6 mA
Output High Voltage (Voh)	4.5 2.4			V V	Ioh=-10 μA Ioh=-400 μA

COMMUNICATION LINE DC ELECTRICAL CHARACTERISTICS

Characteristic	Minimum	Typical	Maximum	Units	Condition
Differential Driver Output Voltage			5.0	V	Unloaded See Note 1
RS-422	2.0		5.0	V	R=50 Ω
RS-485	1.5		5.0	V	R=27 Ω
Maximum Receiver Input voltage			±14	V	
ESD Protection		2000		V	

A/D CONVERTER CHARACTERISTICS

Characteristic	Minimum	Typical	Maximum	Units	Condition
Resolution	12			bits	
Linearity Error		±3/4		bits	
Offset and Gain Error		±2		bits	
Voltage Reference	5.5	5.0	4.5	V	VREF is Vcc
Analog Input Range		-0.5 to Vcc+0.05		V	See Note 2
Analog Input Impedance		250k		Ω	See Note 3

250 reads per second straight BASIC - 7000 reads per second straight assembly

Note 1: RS-232A is characterized as a ±5-V bipolar signal (as opposed to RS-232C at ±12 V). Drivers and receivers are actually RS-422 and the interface is an RS-423 connection (single ended to differential). Domino RS-232A Voltage output is 0-5V only.

Note 2: Two diodes are tied to each analog input which will conduct when the input voltage is one diode drop below ground

or one diode drop above Vcc. To achieve absolute 0-5-V input range requires Vcc to be greater than 4.950 V.

Note 3: The ADC input impedance is a function of clock frequency. The sampling frequency of the DOMINO ADC built-in utility results in a typical impedance of 250 kΩ.

1.0 PROGRAMMING CHARACTERISTICS

DOMINO-2 is a complete computer/controller in one tiny package. The embedded BASIC interpreter and firmware provide the user with a direct means to enter and save an autostarting control program without expensive development tools. Such powerful advantages facilitate completing a programming task in record time. You can write, test, and save code in nonvolatile storage directly on DOMINO-2.

The friendly, control-oriented BASIC command set allows easy access to the integrated digital and analog I/O functions. Conversion calculations are a breeze thanks to BASIC's floating-point number crunching. Because of the power of a high-level language such as BASIC, useful programs often take less than a dozen programming statements. Nonetheless, DOMINO-2 has over 30 KB of space reserved for your application code and the utilities. For Application notes, please visit www.micromint.com

Even though DOMINO-2 is optimized for BASIC programs, assembly language programs are easily accommodated as callable routines. A DOMINO-2 application program can be all BASIC, BASIC with callable assembly language routines, or virtually all assembly language with the only BASIC command being an introductory CALL.

DOMINO-2 contains all the communication interface hardware. It can be used standalone to monitor analog and digital inputs and to provide control outputs directly to machine or network interfaces. When connected serially, DOMINO-2 can serve as a remote device, reporting monitored conditions to your PC or receiving commands to control external components. If multiple DOMINO-2s are networked with a master PC or another DOMINO-2, multi-drop units can share information collected throughout the network.

2.0 MEMORY MAP

The 64-KB memory is based on an 8051 microcontroller's memory structure. The upper 32 KB is devoted to ROM and the lower 32 KB to RAM.

2.1 Development Mode

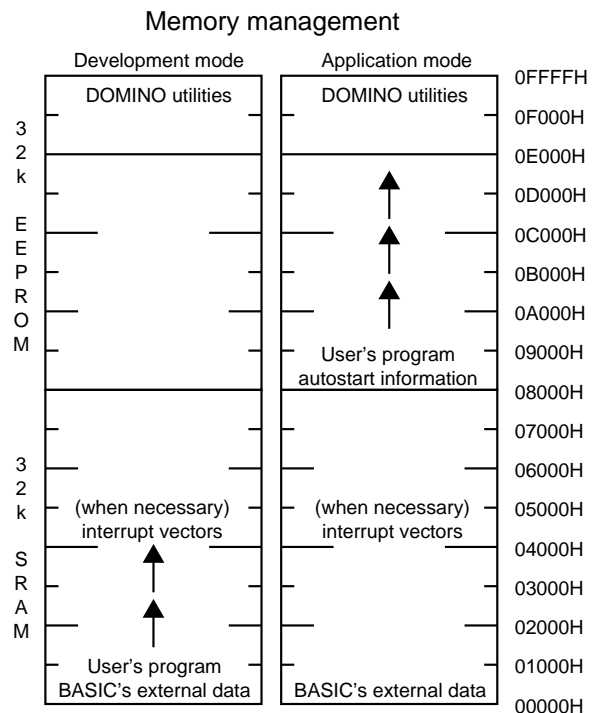
When you're in the development mode, the lower 32 KB of memory is used as temporary storage for BASIC's programs, variables, and jump vectors.

The development mode is used to test and debug BASIC programs. The top of the upper ROM space holds the utilities which are callable functions complementing BASIC's floating-point commands.

2.2 Applications Mode

Finished BASIC programs are saved in the upper 32 KB of nonvolatile ROM space along with the utilities. BASIC programs can be autoexecuted on powerup or reset.

The lower 32 KB of RAM space is used for storage of temporary variables and jump vectors.



3.0 BASIC INSTRUCTION SET

Command	Function
RUN	Execute a program
CONT	Continue after a stop or Control-C
LIST	List program to the console device
LIST#	List program to serial printer port (P1.7)
NEW	Erase the program stored in RAM
NULL	Set null count after carriage return/line feed
RAM	Evoke RAM mode, current program in read/write memory
ROM	Evoke ROM mode, current program in ROM/EPROM
XFER	Transfer a program from ROM/EPROM to RAM
Statement	Function
ASC()	Returns integer of ASCII character
BAUD	Set data-transmission rate for line-printer port
CALL	Call assembly-language program
CHR()	Returns ASCII character of integer
CLEAR	Clear variables, interrupts, and strings
CLEAR\$	Clear stacks
CLEARI	Clear interrupts
CLOCK1	Enable real-time clock
CLOCK0	Disable real-time clock
DATA	Data to be read by READ statement
READ	Read data in DATA statement
RESTORE	Restore READ pointer
DIM	Allocate memory for arrayed variables
DO	Set up loop for WHILE or UNTIL
UNTIL	Test DO loop condition (loop if false)
WHILE	Test DO loop condition (loop if true)
END	Terminate program execution
FOR-TO-{STEP}	Set up FOR...NEXT loop
NEXT	Test FOR...NEXT loop condition
GOSUB	Execute subroutine
RETURN	Return from subroutine
GOTO	GOTO program line number
ON GOTO	Conditional GOTO
ON GOSUB	Conditional GOSUB
IF-THEN-{ELSE}	Conditional test
INPUT	Input a string or variable
LET	Assign a variable or string a value (LET is optional)
ONERR	ONERR or GOTO line number
ONTIME	Generate an interrupt when time is equal to or greater than ONTIME argument; line number is after comma
ONEX1	GOSUB to line number following ONEX1/ when INT1 pin is pulled low
PRINT	Print variables, strings, or literals, P. is shorthand for print
PRINT#	Print to serial printer port (P1.7)
PH0.	Print hexadecimal mode with zero suppression
PH1.	Print hexadecimal mode with no zero suppression
PH0.#	PH0.# to serial printer port (P1.7)
PH1.#	PH1.# to serial printer port (P1.7)
PUSH	Push expressions on argument stack
POP	Pop argument stack to variables
PWM	Pulse-width modulation
REM	Remark
RETI	Return from interrupt
STOP	Break program execution
STRING	Allocate memory for strings
UI1	Evoke user console input routine
UI0	Evoke BASIC console input routine
UO1	Evoke user console output routine
UO0	Evoke BASIC console output routine

Operator	Function
CBY()	Read program memory
DBY()	Read/assign internal data memory
XBY()	Read/assign external data memory
GET	Read console
IE	Read/assign IE register
IP	Read/assign IP register
PORT1	Read/assign I/O port 1 (P1)
PCON	Read/assign PCON register
RCAP2	Read/assign RCAP2 (RCAP2H:RCAP2L)
T2CON	Read/assign T2CON register
TCON	Read/assign TCON register
TMOD	Read/assign TMOD register
TIME	Read/assign real-time clock
TIMER0	Read/assign TIMER0 (TH0:TL0)
TIMER1	Read/assign TIMER1 (TH1:TL1)
TIMER2	Read/assign TIMER2 (TH2:TL2)
+	Addition
/	Division
**	Exponentiation
*	Multiplication
-	Subtraction
.AND.	Logical AND
.OR.	Logical OR
.XOR.	Logical exclusive OR

Stored Constant

PI	PI - 3.1415926
----	----------------

Operators-Single Operand

ABS()	Absolute value
NOT()	One's complement
INT()	Integer
SGN()	Sign
SQR()	Square root
RND	Random number
LOG()	Natural log
EXP()	"e" (2.7182818) to the X
SIN()	Returns the sine of argument
COS()	Returns the cosine of argument
TAN()	Returns the tangent of argument
ATN()	Returns the arctangent of argument

Utility Calls (executed as CALL {address})

PROG	Save the current program in EEPROM
PROG1	Save data-transmission-rate information in EEPROM
PROG2	Save data-transmission-rate information in EEPROM and execute program after reset
PROG3	Save data-transmission-rate information in EEPROM and saves MTOP
PROG4	Save data-transmission-rate information in EEPROM and execute program after reset
ADC	Read 0-5-V input on AD0 or AD1, measurement returned as floating-point value
PWM	Continuous background PWM tasking
FREQ	Measurement of TTL input frequency
PERIOD	Measurement of TTL input period
I ² C	Communication with external I ² C connected coprocessor and externally connected peripheral chips

3.1 Domino Utilities Function Calls

<u>Feature</u>	<u>Function</u>	<u>Call Address</u>
12- bit Analog-Digital Conversion	Single-ended channel 0	0F000H
	Single-ended channel 1	0F008H
	Differential +/-	0F010H
	Differential -/+	0F018H
	Single-ended channel 1 & 0	0F020H
8-Bit Analog-Digital Conversion NOTE: Only available when connected externally	Single-ended channel 0	0F080H
Pulse-Width Modulation	Start PWM0 using TIMER0 and *INT0 ad outputs	0F060H
	Stop PWM0 immediately	0F068H
	Start PWM1 using TIMER1 and *INT1 ad outputs	0F064H
	Stop PWM1 immediately	0F06CH
Period and Frequency	Start measurement on*INT0	0F070H
	Start measurement on *INT1	0F074H
	Retrieve measurement on INT0	0F078H
	Retrieve measurement on INT1	0F07CH
Program the EEPROM (PROGx)	PROG	0FF00H
	PROG1	0FF08H
	PROG2	0FF10H
	PROG3	0FF18H
	PROG4	0FF20H
I ² C Byte Transfer	Retrieve Registered Byte	0F12CH
	Send Byte	0F120H
	Retrieve Byte	0F124H
I ² C Beeper		0F110H
I ² C Keypad	Initialize	0F100H
	Hook into UI0/1 BASIC command	0F108H
I ² C LCD	Initialize	0F030H
	Clear display and home cursor	0F038H
	Display \$(0) string	0F040H
	Hook into UI0/1 BASIC command	0F050H
Utilities Version Number		0FFF0H

4.0 DOMINO-2 FUNCTION CALL PROCEDURES

Micromint has included additional utilities with its built-in BASIC interpreter. Not only do you have the power of a full floating-point BASIC, but you also have extra functions to help make your application extremely easy to produce. The added functions include analog measurement, dual PWM outputs, dual period/frequency input measurements, I²C bus compatibility (e.g., LCD output and keypad input), and program storage in EEPROM for autostarting your application on power-up. These functions are written in assembler to be extremely fast. They are simple to use straight from BASIC.

ADC	read 0–5-V input on AD0 or AD1, measurement returned as floating-point value
PWM	continuous background PWM tasking
FREQ	measurement of TTL input frequency
PERIOD	measurement of TTL input period
PROG1–4	autoexecutable program storage into nonvolatile EEPROM
I ² C	communication with external I ² C peripherals

These function calls are loaded into the EEPROM above BASIC program storage by a utility loader. The DOMINO-2 firmware is preloaded at the factory prior to shipment. Should it be accidentally erased or need to be revised, it can be reprogrammed using the bootstrap loader diskette included in the DOMINO-2 development software package.

DOMINO-2 is potentially reprogrammable even while soldered in an end-use application. This reprogramming reduces obsolescence, making it possible for a user to upgrade current DOMINO-2 stock with the latest enhancements.

4.1 12-Bit Analog-Digital Conversion

Syntax: CALL {address}
POP {variable}

Function: The CALL initiates an analog-to-digital conversion. The result is presented on the stack to be POPed by the user.

Mode: Command, Run

Use: [single-ended channel 0]
CALL 0F00H
POP {variable}
[single-ended channel 1]
CALL 0F008H
POP {variable}
[differential +/-]
CALL 0F010H
POP {variable}
[differential -/+]
CALL 0F018H
POP {variable}
[single-ended channel 1 & 0]
CALL 0F020H
POP {variable},{variable}

Description: The processor's Port1 pins (P1.7 *CS, P1.6 Data, and P1.5 CLK) are used to access either the internal LTC1298 (DOMINO-2A) or an externally connected LTC1298 (DOMINO-2). The LTC1298 offers a number of different connection configurations. Two ADC input channels are available when each is a single-ended measurement (referenced to ground). Alternatively, these channels can be used as a single differential input (neither is ground referenced but there is no greater than 5 V between them). Channel 0 is +input, channel 1 is -input.

Related Topics: 8-bit A/D Conversion

Error Presentation: No errors presented. A CALL made to a nonexistent ADC still returns a value on the stack, albeit one of no meaning.

Example:

```
10 PRINT "This program prints an A/D conversion"
20 PRINT " from two single-ended inputs: Channel
   1 & 0"
30 INPUT "Measure and enter your VCC voltage
   (e.g., 5.12)" P
40 CALL 0F020H: REM THE FUNCTION CALL
50 POP V1.V0: REM GETTING THE RESULTS
60 PRINT USING(0),"Channel 1's conversion count
   is",V1
70 PRINT " and the calculated voltage is",
80 PRINT USING(0),"V1 * (P/4096)," volts"
90 PRINT USING(0),"Channel 0's conversion count
   is",V0
100 PRINT " and the calculated voltage is"
110 PRINT USING(0),"V0 * (P/4096)," volts"
120 PRINT "Hit a <cr> to make another conversion"
    : PRINT
130 IF (GET=0) THEN GOTO 130 ELSE GOTO 40
```

READY
>RUN

Program Output:

```
This program prints an A/D conversion
from two single-ended inputs: Channel 1 & 0
Measure and enter your VCC voltage (e.g., 5.12) ?
4.95
Channel 1's conversion count is 254
and the calculated voltage is 0.310 volts
Channel 0's conversion count is 1259
and the calculated voltage is 1.521 volts
Hit a <cr> to make another conversion
```


4.2 8-Bit Analog-Digital Conversion

Syntax: CALL {address}
POP {variable}

Function: The CALL initiates an analog-to-digital conversion. The result is presented on the stack to be POPed by the user.

Mode: Command, Run

Use: [single-ended channel 0]
CALL 0F080H
POP {variable}

Description: The processor's Port1 pins (P1.7 *CS, P1.6 Data, and P1.5 CLK) are used to access an externally connected ADC0831 (DOMINO-2). The ADC0831 offers a single-ended measurement (referenced to ground).

Related Topics: 12-bit A/D Conversion

Error Presentation: No errors presented. A CALL made to a nonexistent ADC still returns a value on the stack, albeit one of no meaning.

Example:

```
10 PRINT "This program prints an A/D conversion"
20 PRINT " from a single-ended input on Channel
   0"
30 INPUT "Measure and enter your VCC voltage
   (e.g., 5.12)" P
40 CALL 0F080H: REM THE FUNCTION CALL
50 POP V0: REM GETTING THE RESULTS
60 PRINT USING(0),"Channel 0's conversion count
   is",V0
70 PRINT " and the calculated voltage is",
80 PRINT USING(0.###), V0 * (P/256)," volts"
90 PRINT "Hit a <cr> to make another conversion"
   : PRINT
100 IF (GET=0) THEN GOTO 100 ELSE GOTO 40

READY
>RUN
```

Program Output:

```
This program prints an A/D conversion
 from a single-ended input on Channel 0.
Measure and enter your VCC voltage (e.g., 5.12) ?
4.95.
Channel 0's conversion count is 54
 and the calculated voltage is 1.044 volts.
Hit a <cr> to make another conversion
```

4.3 Pulse-Width Modulation (See p. 18 for hardware PWM)

Syntax: PUSH {On time},{Off time},{Duration}
CALL {address}

where variable:

(On time) = integer 150–65535 counts
(Off time) = integer 150–65535 counts
(1 count = 1.085 μ s)
(Duration) = integer 0–255 cycles
(0 = continuous)

and given that:

1–99% duty cycle pulses up to 60 Hz
50% duty cycle pulses up to 3 kHz

Function: Defines the on time (high), off time (low), and duration (# of complete cycles) for a PWM output signal. It also starts the PWM output. A duration of zero means continuous output. Two separate PWM outputs are available *INT0 uses TIMER0 and T1 uses TIMER1.

WARNING: Using PWM0 disables all other functions using Timer0 and INT0. Using PWM1 disables all other functions using Timer1.

BASIC commands using:

TIMER0: CLOCK1
TIMER1: PWM, LIST#, PRINT#
INTERRUPT 0: none

Mode: Command, Run

Use: [start PWM0 using TIMER0 and *INT0 as output]
PUSH 500,1500,0
CALL 0F060H

[stop PWM0 immediately]
CALL 0F068H

[start PWM1 using TIMER1 and T1 as output]
PUSH 500,1500,0
CALL 0F064H
[stop PWM1 immediately]
CALL 0F06CH

Description: Using the PWM function requires MTOP to be set to 3FFFH (although the function call sets this, the user should be aware that any variable used prior to this call is destroyed unless MTOP is preset to 3FFFH at the beginning of a program).

TIMER interrupt vector locations (400BH–400DH and 401BH–401DH) and on time, off time, and duration values storage locations (4200H–420BH) are set up in RAM. The PWM function call sets up the TIMER counts alternating between the on-time value and the off-time value on each TIMER overflow until the duration value has been decreased to zero.

A separate function call can be made at any time to immediately shut down the PWM. Each on- and off-time count defined is the number of 1.085- μ s tics the routine delays before changing state. The minimum count is 150 ($150 \times 1.085 \mu\text{s}$) or 163 μs . The max count is 65,535 or 71 ms.

Related Topics: PWM (BASIC command). The BASIC-52 Interpreter's PWM command halts execution of the BASIC program while it is being executed. PWM0 and PWM1 function calls do NOT halt the execution of the BASIC program, but it becomes a background task.

Error Presentation: No error are reported although any BASIC command which uses the timers is disabled (see Function description above).

Example: This example sets up both PWM outputs with continuously varying 1–99% duty cycles.

```

10  FOR Y=150 TO 14700 STEP 300
20  PUSH Y
30  PUSH 15000-Y
40  PUSH 0
50  CALL 0F060H
60  PUSH 15000-Y
70  PUSH Y
80  PUSH 0
90  CALL 0F064H
91  FOR Z=1 TO 50: NEXT Z
100 NEXT Y
110 FOR Y=14700 TO 150 STEP -300
120 PUSH Y
130 PUSH 15000-Y
140 PUSH 0
150 CALL 0F060H
160 PUSH 15000-Y
170 PUSH Y
180 PUSH 0
190 CALL 0F064H
191 FOR Z=1 TO 50: NEXT Z
200 NEXT Y
210 GOTO 10

READY
>RUN

```

4.4 Period and Frequency

Syntax: CALL {address} [Start measurement]
 CALL {address} [Retrieve result]
 POP {variable}

where variable:

(period count) = integer 0–65535
 (0 = measurement started)
 (1 = measurement in process)
 (2 = overflow occurred—signal too slow)
 (60–65535 = counts between negative edges)
 (1 count = 1.085 μs)
 (period = 65 μs –71 ms)
 (frequency = 15 kHz–15 Hz)

Function: The start measurement function call sets up the edge-triggered input interrupts and timers used to measure the period between two successive input edges. Two separate input signals can be measured. Input *INT0 uses interrupt 0 and timer0 and input *INT1 uses interrupt 1 and timer1. **WARNING:** Using either of these inputs disables any other function using the interrupts or timers. The timers and interrupts may again be used after the function calls are complete. BASIC commands using:

```

TIMER0: CLOCK1
INTERRUPT0: none
TIMER1: PWM
LIST#, PRINT# INTERRUPT1: ONEX1

```

Mode: Command, Run

Use: [Start a measurement on input *INT0]
 CALL 0F070H

[Start a measurement on input *INT1]
 CALL 0F074H

[Retrieve a measurement on input *INT0]
 CALL 0F078H
 POP P

[Retrieve a measurement on input *INT1]
 CALL 0F07CH
 POP P

Description: Using the PERIOD/FREQUENCY function requires MTOP to be set to 3FFFH (although the function call sets this, the user should be aware that any variable used prior to this call is destroyed unless MTOP is preset to 3FFFH at the beginning of a program). External interrupt vector locations (4003H–4005H and 4013H–4015H), TIMER interrupt vector locations (400BH–400DH and 401BH–401DH) and period count storage locations (420CH–420FH) are set up in RAM. The start period measurement function call initializes the INTERRUPT and TIMER. The retrieve measurement function call passes the measurement status back to the user via the stack. The status is indicated as follows:

POPped Value	Meaning
0	waiting for input
1	measurement in process
2	overflow (input too slow or nonexistent)
other	counts in 1.085 μ s intervals.

Related Topics: none

Error Presentation: No errors reported (see Function description above).

Example:

```

10 PRINT "This program lets you use the
   frequency function"
20 PRINT "Apply the TTL frequency to pin *INT0
   and/or *INT1"
30 CALL OF070H: REM PERIOD COUNT ON *INT0
   FUNCTION CALL
40 CALL OF074H: REM PERIOD COUNT ON *INT1
   FUNCTION CALL
50 CALL OF078H: REM RETRIEVE COUNT ON *INT0
   FUNCTION CALL
60 POP PC0: REM GET THE PERIOD COUNT
70 IF (PC0=2) THEN PRINT "The frequency is too
   low on *INT0": GOTO 130
80 IF (PC0=0.OR.PC0=1) THEN GOTO 50
90 P=PC0*1.085: REM PERIOD TIME CALCULATION FROM
   COUNT VALUE
100 PRINT "The period on *INT0 is",P,"  $\mu$ s. The
   frequency is",
110 F=1/P*1000000: REM FREQUENCY CALCULATION FROM
   PC VALUE

```

```

120 PRINT F," Hz"
130 CALL OF07CH: REM RETRIEVE COUNT ON *INT1
   FUNCTION CALL
140 POP PC1: REM GET THE PERIOD COUNT
150 IF (PC1=2) THEN PRINT "The frequency is too
   low on *INT1": GOTO 210
160 IF (PC1=0.OR.PC1=1) THEN GOTO 130
170 P=PC1*1.085: REM PERIOD TIME CALCULATION FROM
   COUNT VALUE
180 PRINT "The period on *INT1 is",P,"  $\mu$ s. The
   frequency is",
190 F=1/P*1000000: REM FREQUENCY CALCULATION FROM
   PC VALUE
200 PRINT F," Hz"
210 PRINT "Hit a <cr> to take another sample"
   :PRINT
220 IF (GET=0) THEN 220 ELSE GOTO 30
READY
>RUN

```

Program Output:

This program lets you use the frequency function
 Apply the TTL frequency to pin *INT0 and/or *INT1
 The frequency is too low on *INT0
 The period on *INT1 is 2164.575 μ s. The frequency is 461.9 Hz
 Hit a <cr> to take another sample

4.5 Program EEPROM (PROGx)

Syntax: CALL {address}

Function: The BASIC program residing in RAM and the appropriate header information (autostarting, baud rate, and MTOP) is stored in EEPROM.

Mode: Command

Use: [PROG]

```

[PROG1] CALL OFF00H
[PROG2] CALL OFF08H
[PROG3] CALL OFF10H
[PROG4] CALL OFF18H
        CALL OFF20H

```

Description: These function calls act just like the BASIC-52 PROG commands. The BASIC commands are written for EPROM. The EEPROM used here requires a different programming algorithm. The PROG function call replaces the BASIC PROG command and saves only the program. The remaining PROGx function calls are similar in function to the BASIC commands, but the function calls all save the program and the startup characteristics in a single call.

Related Topics: PROG, PROG1, PROG2, PROG3, and PROG4 (all BASIC commands)

Error Presentation:

ABORT, PROGRAMMING ERROR!
[EEPROM life exceeded]
ABORTED, ILLEGAL ACCESS ATTEMPT!
[storage space exceeded]
ABORTED, UNKNOWN RESULT CODE!
[unknown error]
ABORTED, NOTHING TO PROGRAM!
[no program in RAM]

Example:

[Type in your program:]

```
10 PRINT "Hello World!"
```

[Type RUN to verify it executes properly:]

```
RUN
```

Hello World!

>READY

[Now type the function call for PROG2:]

```
CALL 0FF10H
```

[you should see:]

```
STORING PROGRAM...
```

```
PROGRAM STORED!
```

[Whenever the power is disconnected and reconnected you should see:]

Hello World!

4.6 I²C Byte Transfers

Syntax: [send registered BYTE]

```
PUSH {slave address * 100H + slave register}  
PUSH {8-bit value}  
CALL {address}  
POP {16-bit value}
```

[retrieve registered BYTE]

```
PUSH {slave address * 100H + slave register}  
CALL {address}  
POP {16-bit value}
```

[send BYTE]

```
PUSH {slave address * 100H + 8-bit value}  
CALL {address}  
POP {16-bit value}
```

[retrieve BYTE]

```
PUSH {slave address * 100H}  
CALL {address}  
POP {16-bit value}
```

Function: Communication is attempted with an I²C device. An 8-bit value is passed to and from the device.

Mode: Command, Run

Use: where

A=slave address

R=slave register

V=value to send

C=value retrieved

[send registered BYTE]

```
PUSH A*100H+R,V
```

```
CALL 0F128H
```

```
POP C
```

[retrieve registered BYTE]

```
PUSH A*100H+R
```

```
CALL 0F12CH
```

```
POP C
```

[send BYTE]

```
PUSH A*100H+V
```

```
CALL 0F120H
```

```
POP C
```

[retrieve BYTE]

```
PUSH A*100H
```

```
CALL 0F124H
```

```
POP C
```

Description: The address and register of the I²C slave device is passed on the stack. If an 8-bit value is to be sent, it too must be pushed onto the stack. A call is then made to send a message using the I²C bus (P1.7 CLK and P1.6 DATA). The routine returns a 16-bit value to the user on the stack. The upper byte of the returned value is zero (000xxH) if the transfer was successful. Otherwise, it is set to all 1s (0FFxxH). If the function was to retrieve a byte, it is in the lower 8 bits of the 16-bit return.

Related Topics: I²C Beeper, I²C Keypad, and I²C LCD. See Appendix 2 for schematics.

Error Presentation: The upper 8 bits of the received byte are masked to all 1s if the transmission is unsuccessful or 0s if all is OK.

Note: See section 6.0 for using the DOMINO-2 I²C coprocessor.

Example:

```

10 PRINT"Turn ON the beeper"
20 A=046H
30 V=0DFH
40 PUSH A*100H+V
50 CALL 0F120H
60 POP C
70 IF (C<>0) THEN GOTO 180
80 PRINT"Hit a key to turn it OFF"
90 G=GET
100 IF (G=0) THEN GOTO 90

110 V=0FFH
120 PUSH A*100H+V
130 CALL 0F120H
140 POP C
150 IF (C<>0) THEN GOTO 180
160 PRINT"Now it's OFF"
170 END
180 PRINT"Error in I2C communications"
190 END

```

4.6.1 I²C Beeper

This function call assumes you're using a Philips/Signetics PCF8574 I²C 8-bit I/O expander with the slave address 01000110 and piezobeeper on bit 5.

Syntax: CALL {address}

Function: Bit 5 of the slave I/O expander is momentarily set low to produce a short burst from an attached piezo-beeper.

Mode: Command, Run

Use: CALL 0F110H

Related Topics: I²C Keypad, I²C LCD. See Appendix 2 for schematics.

Error Presentation: none

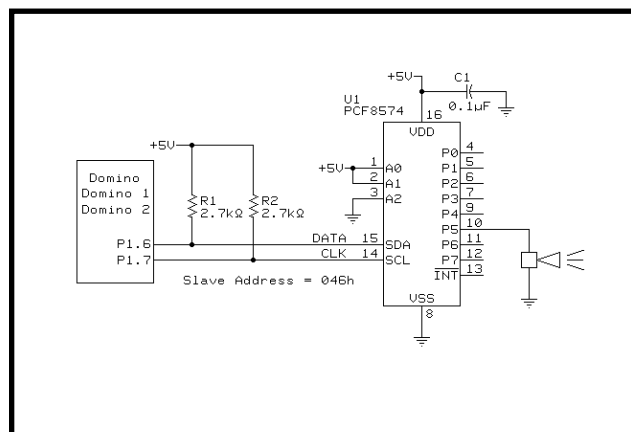
Example:

```

10 PRINT"Beep the beeper"
20 CALL 0F110H
30 PRINT"That's it!"
40 END

```

Description: The pre-assigned I²C slave address 46H is written to with a value of DFH to turn off bit 5. After a short delay, a value of FFH is sent to turn bit 5 back on. The output bit can sink 25 mA of current for, in this case, a piezoelectric beeper.



4.6.3 I²C LCD

This function call assumes you're using a Philips/Signetics PCF8574 I²C 8-bit I/O expander with the slave address 01000010 and output bit 0–3 to the (LM034—4x20) LCD data bits 4–7, output bit 4 to the LCD RS pin, and output bit 5 to the LCD E pin).

Syntax: [initialize]
 CALL {address}
 [clear display & home cursor]
 CALL {address}
 [display \$(0) string]
 CALL {address}
 [display a character]
 CALL {address}
 [hook into UI0/1 BASIC command]
 CALL {address}

Function: The LCD must be initialized through the slave I/O expander. This sets up the LCD in nibble mode with a 4 × 7 character matrix and invisible cursor. The LCD is cleared and the cursor set to row1 column 1. Once initialized, the LCD can be cleared and cursor sent home at any time. The first string, \$(0), can be directed to the LCD without using console redirection (UO1).

If you choose to use console redirection (UO1), the characters are handled one at a time. Console redirection can be invoked once the hooks are installed for the BASIC UI1 command.

Mode: Command, Run

Use: [initialize]
 CALL 0F030H
 [clear display & home cursor]
 CALL 0F038H
 [display \$(0) string]
 CALL 0F040H
 [hook into UI0/1 BASIC command]
 CALL 0F050H

Description: To initialize the LCD, the pre-assigned I²C slave address 42H is used as an output port. The initialization data is sent to the output port to place the LCD in nibble mode with a 4- × 7-character matrix and invisible cursor. The clear display and home cursor function is called to complete the initialization. Clearing the display and homing the cursor can be used any time after the LCD has been initialized. Since the LCD does not clear from the end of a print string to the end of the LCD line, you will find this function call necessary to keep the screen clean.

Displaying a string is easy without console redirection. The first string, \$(0), can be displayed on the LCD by using a simple function call. This displays all characters in the string (normally unprintable characters may be displayed as Kana characters).

Alternatively, and much easier to use, the console output device can be hooked in as the secondary or alternate output device. The use of the BASIC UO1 command redirects all output automatically to the LCD display. Once hooked, characters are printed on a character by character basis. Characters between 20H and 7FH are displayed. Those above 7FH are used as cursor control. A <cr> moves the cursor to the beginning of the next (or first) line. Using the hook function requires MTOP to be set to 3FFFFH (although the function call sets this, the user should be aware that any variable used prior to this call is destroyed unless MTOP is preset to 3FFFFH at the beginning of a program).

Custom console output vector locations (4030H–4032H) and custom list@/print@ vector locations (403CH–403EH) are set up in RAM.

NOTE: LCD output routines are considerably slower than console output, therefore care must be taken when using redirected (UO1) output to the LCD while using the BASIC-52 interpreter's INPUT \$(0) command. Input characters can be lost while the previous character's ECHO is being displayed when you run above 4800 bps. You can either use a data rate of less than 9600 or redirect console output to the primary (serial port) until after the INPUT statement. Then, when input is complete, redirect the console output to the LCD and PRINT \$(0).

Related Topics: I²C BYTE transfers, I²C Keypad, UI0 and UI1 (BASIC commands). See Appendix 2 for schematics.

Error Presentation: none

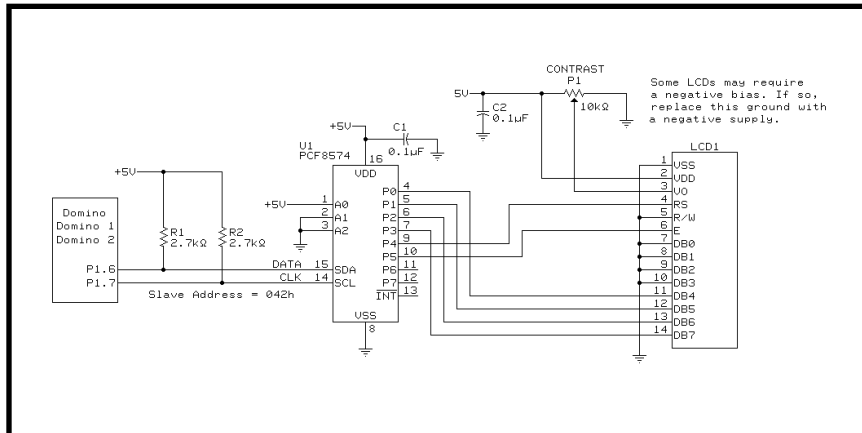
Example:

[direct string output to the LCD]

```
10  MTOP=03FFFF
20  STRING 82,80
30  PRINT"Initialize the LCD"
40  CALL 0F030H
50  PRINT"Now all input strings will be displayed
    on the LCD"
60  INPUT $(0)
70  CALL 0F040H: REM PRINT $(0) TO LCD
80  GOTO 60
```

[redirecting the PRINT command to the LCD as the secondary console output device]

```
10  MTOP=03FFFF
20  STRING 82,80
30  PRINT"Initialize the LCD"
40  CALL 0F030H
50  PRINT"Hook into the secondary console output
    (LCD)"
60  CALL 0F050H
70  PRINT"Now all further output will be displayed
    on the LCD"
80  INPUT $(0)
90  UO1: REM CHANGE TO SECONDARY CONSOLE OUTPUT
    DEVICE
100 PRINT $(0)
110 UO0: REM CHANGE TO PRIMARY CONSOLE OUTPUT
    DEVICE
120 GOTO 70
```



4.7 Utilities Version Number

Syntax: CALL {address}

Function: The CALL initiates a sign-on message displaying the version number of the utilities presently installed.

Mode: Command, Run

Use: CALL 0FFF0H

Description: Version identification, which is embedded in the utilities, is sent to the active console output.

5.0 CONTROLLING I/O BITS DIRECTLY

Since it isn't possible to directly set or reset bits on Port 3 from BASIC-52, it is necessary to call short machine language routines to do the job. The routines consist of three bytes. The first is either a SETB instruction (D2) or a CLR instruction (C2). The second specifies a bit address. Finally, the third is a RET instruction (22).

The following table details the necessary routines for each of the Port 3 bits. The program example shows how to insert the routines into memory from BASIC-52 and how to call them.

NOTE: All data in BASIC-52 must begin with a numeric value or else it is interpreted as a variable. (ex: xby(3200h) = 0D2)

Pin	Bit	Name	To Set	To Clear
P3.0	B0	RxD	D2 B0 22	C2 B0 22
P3.1	B1	TxD	D2 B1 22	C2 B1 22
P3.2	B2	Int 0	D2 B2 22	C2 B2 22
P3.3	B3	Int 1	D2 B3 22	C2 B3 22
P3.4	B4	T0	D2 B4 22	C2 B4 22
P3.5	B5	T1	D2 B5 22	C2 B5 22

Example:

The following code is an example for using INT 1 (P3.3) as an output bit.

```
100  MTOP=31000: REM Set MTOP Lower
110  XBY (32000)=0D2H: XBY(32001)=0B3h:
      XBY(32002)=022H
120  REM Put INT 1 Set Program at 32000
```

```
150  XBY(32100)=0C2H: XBY(32101)=0B3H:
      XBY(32102)=022H
160  REM Put INT 1 Reset Program at 32100
200  Call 32000: REM INT 1 On
210  Call 32100: REM INT 1 Off
222  Goto 200
```

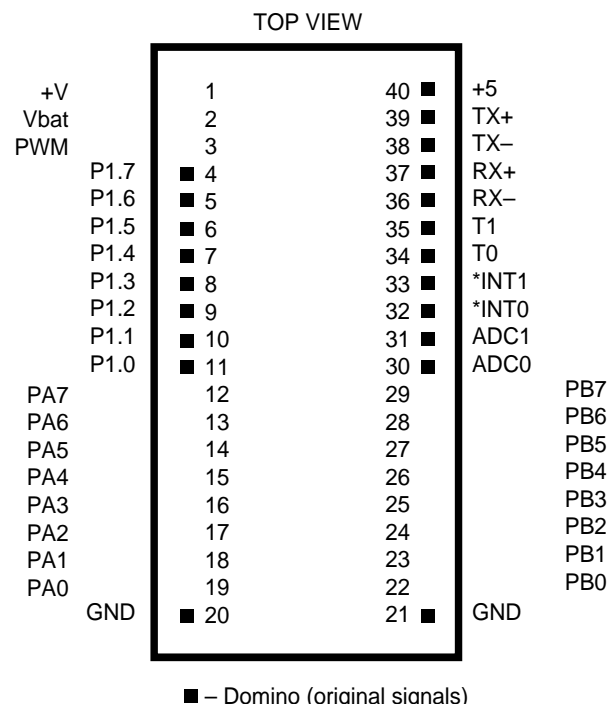
Note: RAM locations and assembly code can be expressed in decimal, hex, or both.

6.0 I/O COPROCESSOR

DOMINO-2 is equivalent to the original DOMINO-52/52A plus additional I/O provided by an onboard I/O coprocessor. The I/O coprocessor is connected to the BASIC processor via the I²C bus. It provides 16 bits of high current drive digital I/O, a hardware background PWM, and a RTClock/calendar. The coprocessor can maintain the correct time and date even if main power is removed by connecting an auxiliary battery source to the Vbat input (Pin 2). Each of the 16 digital I/O pins can be individually programmed as either input or output and they have the capability to directly drive LEDs either sinking or sourcing current. Port A & B can only sink or source up to a combined total of 200mA.

DOMINO-2's coprocessor is physically connected to the BASIC processor's I²C I/O lines (P1.7 and P1.6). These two pins may also be used to add external I²C devices (see section 4.6). With the exception of I²C expansion peripherals, **these two lines should not be used for direct I/O**. Circuitry connected to these lines may interfere with the coprocessor.

Coprocessor communications are handled as an I²C register write or register read. The coprocessor is defined as address 20H. There are 51 8-bit registers which can be written to or read from. Each has a distinct address and function. They are defined below.



6.1 DOMINO-2 Coprocessor Register Definitions

Register # hex dec.	Name	Function	Register # hex dec.	Name	Function
30H 48	PA_DIR	Direction Control Register for PORT A	55H 85	PA_OUT5	PORT A Output Bit 5
31H 49	PA_IN	PORT A Input Port	56H 86	PA_OUT6	PORT A Output Bit 6
32H 50	PA_OUT	PORT A Output Port	57H 87	PA_OUT7	PORT A Output Bit 7
33H 51	PB_DIR	Direction Control Register for PORT B	58H 88	PB_OUT0	PORT B Output Bit 0
34H 52	PB_IN	PORT B Input Port	59H 89	PB_OUT1	PORT B Output Bit 1
35H 53	PB_OUT	PORT B Output Port	5AH 90	PB_OUT2	PORT B Output Bit 2
40H 64	PA_IN0	PORT A Input Bit 0	5BH 91	PB_OUT3	PORT B Output Bit 3
41H 65	PA_IN1	PORT A Input Bit 1	5CH 92	PB_OUT4	PORT B Output Bit 4
42H 66	PA_IN2	PORT A Input Bit 2	5DH 93	PB_OUT5	PORT B Output Bit 5
43H 67	PA_IN3	PORT A Input Bit 3	5EH 94	PB_OUT6	PORT B Output Bit 6
44H 68	PA_IN4	PORT A Input Bit 4	5FH 95	PB_OUT7	PORT B Output Bit 7
45H 69	PA_IN5	PORT A Input Bit 5	60H 96	PWM_EN	Start/Stop PWM Output
46H 70	PA_IN6	PORT A Input Bit 6	61H 97	PWM_P	PWM Period Value
47H 71	PA_IN7	PORT A Input Bit 7	62H 98	PWM_D	PWM Duty Cycle Value
48H 72	PB_IN0	PORT B Input Bit 0	63H 99	PWM_PRE	PWM Period Prescale Value
49H 73	PB_IN1	PORT B Input Bit 1	70H 112	TIME_EN	Start/Stop Real Time Clock/Calendar
4AH 74	PB_IN2	PORT B Input Bit 2	71H 113	TIME_S	Seconds Register
4BH 75	PB_IN3	PORT B Input Bit 3	72H 114	TIME_MI	Minutes Register
4CH 76	PB_IN4	PORT B Input Bit 4	73H 115	TIME_H	Hours Register
4DH 77	PB_IN5	PORT B Input Bit 5	74H 116	TIME_WD	Day of the Week Register
4EH 78	PB_IN6	PORT B Input Bit 6	75H 117	TIME_DA	Day of the Month Register
4FH 79	PB_IN7	PORT B Input Bit 7	76H 118	TIME_MO	Month of the Year Register
50H 80	PA_OUT0	PORT A Output Bit 0	77H 119	TIME_Y	Year Register
51H 81	PA_OUT1	PORT A Output Bit 1	7FH 127	VER	Software Version Number
52H 82	PA_OUT2	PORT A Output Bit 2			
53H 83	PA_OUT3	PORT A Output Bit 3			
54H 84	PA_OUT4	PORT A Output Bit 4			

6.2 Requesting a Register Value from the Coprocessor

To get a value from any of the coprocessor's registers, use the I²C retrieve registered byte command. A typical BASIC-52 program is as follows (see Section 4.6 for I²C generic command definition):

```
PUSH A*100H+R ; parameter setup
CALL 0F12CH   ; I2C function
POP C         ; parameter return
```

where:

A=20H the address of the coprocessor
 R= the register number you wish to retrieve
 C= the value returned by the CALL

If you want to get the version number use R=7FH and A=20H.

```
PUSH 207FH
CALL 0F12CH
POP C
```

If the variable 'C' has a value greater than 255, then it means there has been a communications error with the coprocessor. A value less than 255 represents the present Version number.

6.3 Updating a Register Value to the Coprocessor

To place a new value into any of the coprocessor's registers use the I²C send registered byte command:

```
PUSH A*100H+R,V ; parameter setup
CALL 0F128H      ; I2C function
POP C            ; parameter return
```

where:

A=20H the address of the coprocessor
R= the register number you wish to send to
V=the value you wish to send
C= a value returned by the CALL

If you want to Start the RTClock/calendar use R=70H, V=1, and A=20H.

```
PUSH 2070H,1
CALL 0F128H
POP C
```

If the variable C is greater then 255 then there has been a communications error.

6.4 Digital I/O

Domino-2's coprocessor adds 16 bits of digital I/O to the BASIC processor's original 14 TTL I/O bits (12 bits + 2 I²C lines). These 16 bits are treated as two 8-bit ports. Each bit has a direction control associated with it. The eight PA_DIR bits define directions for PORT A and the eight PB_DIR bits define directions for PORT B. When a direction control bit is set to a 1, that port bit is defined as input. When a direction control bit is set to a 0, that port bit is defined as an output. The default directions at power up is for all PORT A and PORT B bits to be defined as inputs. When a bit direction is set as input, writing to it as an output bit will do nothing. When a bit is set as an output, reading from it as an input will merely reflect the logic state of its current output condition.

The logic state of an output pin can be changed by either writing a full byte to the port register or a single bit to a bit

register. If PA0 (Pin 19) was configured as an output and you wanted to set it to a logic high state, you can either write a full byte PA_OUT with the least significant bit (LSB) set to a 1 (xxxxxxx1) or, you can write a 1 to PA_OUT0. PA_OUT and PB_OUT require a byte value (00H–0FFH, 0–255) while PA_OUT0–7 and PB_OUT0–7 require a bit value (0–1).

When configured as inputs, an I/O pin's logic state can be retrieved with either a port-wide read or bit read. To determine whether PB0 (Pin 22) is high or low, you can either read the PB_IN register where its least significant bit (LSB) reflects the state of the PB0 pin or, you can read PB_IN0 directly. PA_IN and PB_IN retrieve byte-wide values (00H–FFH, 0–255) while PA_IN0–7 and PB_IN0–7 retrieve bit values (0–1).

6.5 PWM

The hardware PWM output uses four registers. The PWM_EN register is the ON/OFF switch for the PWM function. A 1 turns the function ON and a 0 turns the function off. The other three registers control the PWM timing. The PWM period (or frequency) is based on an 8-bit clock value (PWM_P) and a prescale divisor (PWM_PRE). The PWM prescale divisor (PWM_PRE) is set to values of 0, 1, or 4. A 0 means divide by 1; a 1 means divide by 4; and, a 2 means divide by 16.

Initialize the co-processor's register.

First the period in μs = (PWM_P+1) * (prescale value of PWM_P) = 100 * 4 = 400 μs
so set the PWM_P register with a value of 99,

```
PUSH 2061H,99
CALL 0F128H
POP C
```

and the PWM_PRE with a value of 1 (1 represents a prescale value of 4),

```
PUSH 2063H,1
CALL 0F128H
POP C
```

now the duty cycle in percent = $100 * \text{PWM_D} / (\text{PWM_P}+1)$
= $100 * 50 / 100 = 50\%$

```
PUSH 2062H,50
CALL 0F128H
POP C
```

Finally, the output can be enabled by setting the PWM_EN register to 1:

```
PUSH 2060H,1
CALL 0F128H
POP C
```

The period register (PWM_P) can be any 8-bit value (00H–0FFH, 0-255). The period is therefore (PWM_P+1) * prescale value of (PWM_PRE) in uS or if PWM_P=99 and PWM_PRE=1 then the period = $100 \times 4 = 400 \mu\text{s}$. The frequency is therefore $1/\text{period}$ or $1/0.0004 = 2.5 \text{ kHz}$.

The duty cycle of this period is set using the PWM_D register. The default duty cycle is 50%. The amount of time the output stays high during the period is the ratio of the PWM_D/(PWM_P+1). If a 25% duty cycle is needed the PWM_D register is written with a value of 25 because $25/(99+1)=0.25$. The PWM output can be enabled or disabled at any time. Changing the PWM_D register will not alter the period (or frequency) of the PWM output.

6.6 Real-Time Clock/Calendar

The coprocessor has a built in real time clock/calendar. It can be battery-backed with 3 V applied to the Vbat input (Pin 2). An External diode is needed, please refer to Errata sheet for details. The clock is enabled by writing a 1 to the TIME_EN register. The time and date will continue to increment as long as it is powered and enabled. The seven time registers can be read from or written to at any time.

The TIME_S and TIME_MI registers holds the seconds and minutes (0-59). The TIME_H contains the hours (0-23).

The day of the week is in the TIME_DW register (1=SUN-7=SAT). The TIME_DA register holds the day of the month (1–31). The month is presented in the TIME_MO (1–12) register while the TIME_Y register has the year (00–99, with auto rollover to 00 after 99). Hours are required to be in 24 hour format. Conversion and/or display in AM/PM format should be done in the BASIC application program.

7.0 UPDATING THE DOMINO-2 UTILITIES

The utilities reside in the uppermost portion of the memory map. The utilities are placed in nonvolatile memory so they remain along with your saved autostarting BASIC program, even after power has become disconnected. The user can take advantage of updated utilities (when available) by simply re-loading them. This is a two-step process.

First, a utilities loader program (LOADUTIL.BAS) is entered into DOMINO-2. When this program is run, you are prompted to download the actual utilities hex file (UTIL_XXX.HEX). The LOADUTIL.BAS program reads in each paragraph of the hex file, converting and storing it in RAM. When the hex file has

been read, "load successful," "call address = xxxx," and "total checksum = xxxxx" messages are displayed. The total checksum should match the one included in the UTIL_100.DOC file. This file also contains any last-minute information you should be aware.

Second, if you have verified that all is correct, you may transfer the utilities using the direct command "CALL xxxx" as displayed in the above message. You get transfer status and a sign-on banner when the utilities have been transferred into nonvolatile EEPROM.

8.0 GETTING STARTED

Although you can use any communication software with DOMINO-2, Host-52 is a convenient and friendly interface between your PC and DOMINO-2. Host-52 can be used on any DOS-compatible PC with 640 KB of conventional memory. To use Host-52 with DOMINO-2, you need two serial ports. COM1 for the serial connection to DOMINO-2 and COM2 for your serial mouse. (If you use COM1 for your serial mouse, you may select an alternate COM port for the DOMINO-2 through the Serial Option of the Main Menu.)

Connect the DOMINO-2 hardware to the PC's serial port and turn on the power to the DOMINO-2. At this point, unless you already have an autostart program in DOMINO-2, it waits to receive a space character. (If you are using a simple comm program like Procomm to communicate with DOMINO-2, remember that DOMINO-2 sets the baud rate when a space character is entered. Any other entry confuses DOMINO-2. You also need to power DOMINO-2 off and on again if the first character DOMINO-2 receives is NOT a space.)

From the DOS command line, type in Host-52 from the installed directory. Host-52 sets up the screen into windows. The top window is the editing window where you input and revise your programs. The middle window is the console output window where you see DOMINO-2's output. The narrow bottom window is the console input window where you can type direct commands to DOMINO-2. You can activate either the editor (top window) or the console (bottom two windows) by clicking on them with the mouse.

Host-52 automatically sends out a space character in an attempt to make contact with the DOMINO-2. You receive an OK message if all is well.

Click on the top window. Host-52 automatically numbers your BASIC program's lines. Enter this single line where Host-52 has entered the line number 10.

```
10 PRINT"Hello World"
```

Now click on the console window, then the PROGRAM item of the menu bar, and then on SEND ALL. Host-52 sends the BASIC code from its editor to DOMINO-2.

Click on the RUN item on the menu bar and then on START. Host-52 passes the run command to DOMINO-2 and your program executes (out of RAM).

```
Hello World  
READY
```

```
>
```

You can start the program from the console input window by typing:

```
RUN<cr>
```

```
Hello World
```

```
READY  
>
```

This can be saved as an autostart program by typing:

```
CALL OFF20H<cr>
```

```
SAVING PROGRAM...  
PROGRAM SAVED!
```

Remove power from the DOMINO-2 and then power it back up. The program automatically runs.

```
Hello World
```

```
READY  
>
```

Please read the *Host-52 Develop System for BASIC-52 CPUs* for complete information on using Host-52, the *BASIC-52 Programming* for more on BASIC's command syntax, and this manual for more on using the DOMINO and DOMINO-2 Utilities.

Devices sold by Micromint are covered by the warranty and patent indemnification provisions appearing in its Terms of Sale only. Micromint makes no warranty, express, statutory, implied, or by description regarding the information set forth herein or regarding the freedom of the described devices from patent infringement. Micromint makes no warranty of merchantability or fitness for any purposes. Micromint reserves the right to discontinue production and change specifications and prices any time and without notice. This product is intended for use in normal commercial applications. Applications requiring extended temperature and unusual environmental requirements, or applications requiring high reliability, such as military, medical life support or life-sustaining equipment, are specifically **not** recommended without additional processing by Micromint for such application.

APPENDIX 1.0

1.1 Sample Application: Communications

DOMINO-2 can communicate with other serial devices at up to 19,200 bps. It can be connected in one of three configurations: RS-232A, RS-422, or RS-485. DOMINO-2's RS-232A output can be used with most full-duplex PC-type serial devices which normally handle RS-232C provided they can reconcile receiving the lower-voltage transmit level of RS-232A. This three-wire (Tx/Rx/GND) RS-232A connection is created by using the RS-422 input receivers as simple level-shifting inverters as shown in Figure 1. RS-422 is an alternate full-duplex connection which uses two twisted-pair transmission lines (i.e., Tx+/Tx-/Rx+/Rx-) offering long

transmission paths and noise-cancelling techniques. This distance is typically 4000'. This connection is shown in Figure 2.

RS-485 is similar to RS-422 with the exception that it uses a single twisted pair in a half-duplex arrangement (i.e., +/-). This means data transmissions must use the same twisted-pair path to travel in both directions, requiring a simple protocol of only one unit seizing the transmission pair at a time while all others listen. This connection is shown in Figure 3.

Figure 1—Typical RS-232A connections

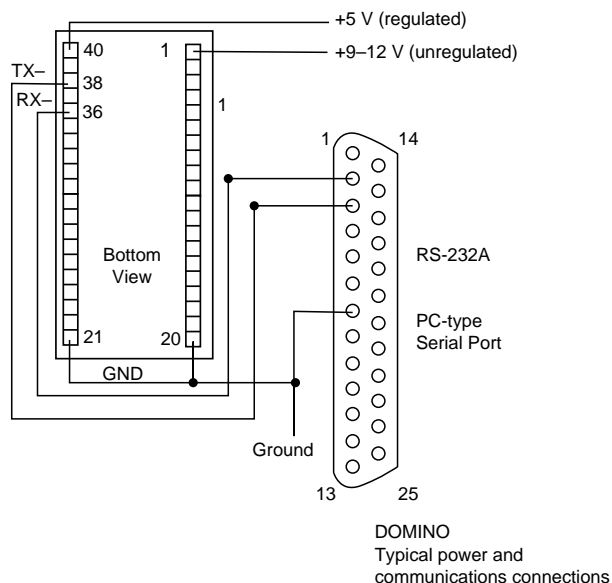
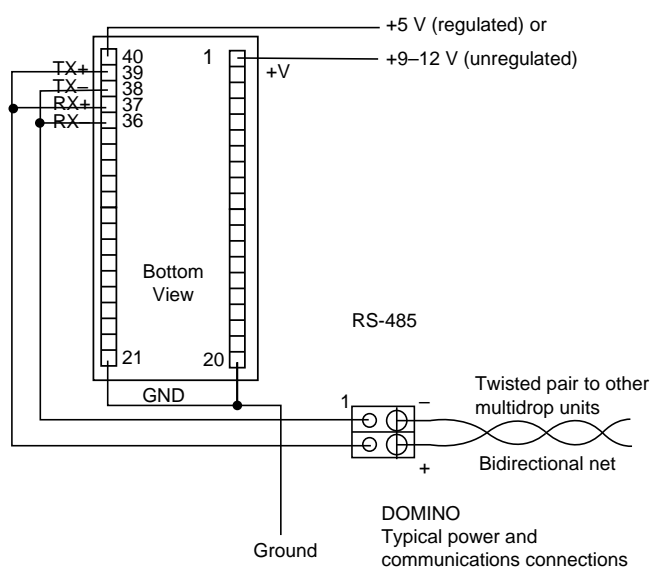
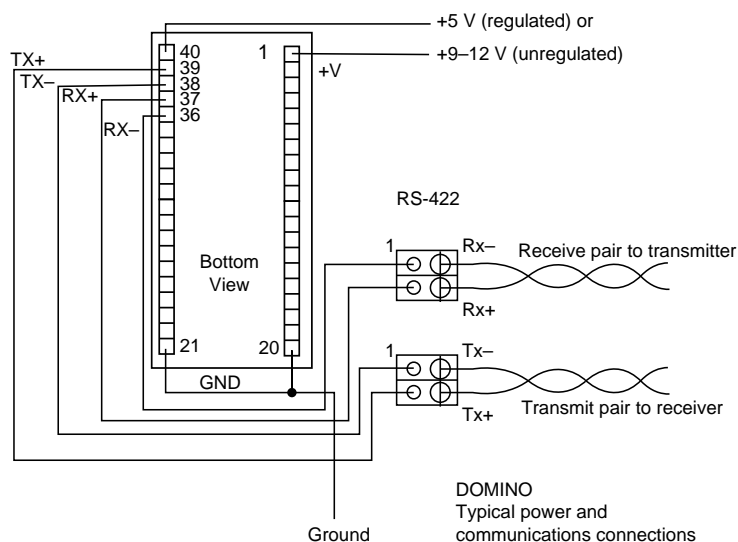


Figure 3—Typical RS-485 connections



Note: RS-485 requires master-slave protocol and direction control

Figure 2—Typical RS-422 connections



1.2 Sample Application: Analog Input Measurement

The DOMINO-2A contains an optional 2-channel, 12-bit A/D converter. The converter is a Linear Technology LTC1298. It is mounted internally in the A version, but can be externally connected to a regular parallel I/O model DOMINO-2. Both DOMINO-2 and DOMINO-2A firmware support ADC calls for 8-bit (ADC0832) and 12-bit (LTC1298) dual-channel ADC devices.

In both applications, the ADC chip is connected to P1.7, P1.6, and P1.5 as described in the pinout listing. When an ADC is connected, these port lines share functions. The user must take care not to confuse functions with random outputs to these lines. The example below shows ADC connections for using DOMINO-2A with the optional internal ADC or DOMINO-2 with an external ADC attached.

Figure 1—Typical connections for using DOMINO-2 with a user-supplied external ADC

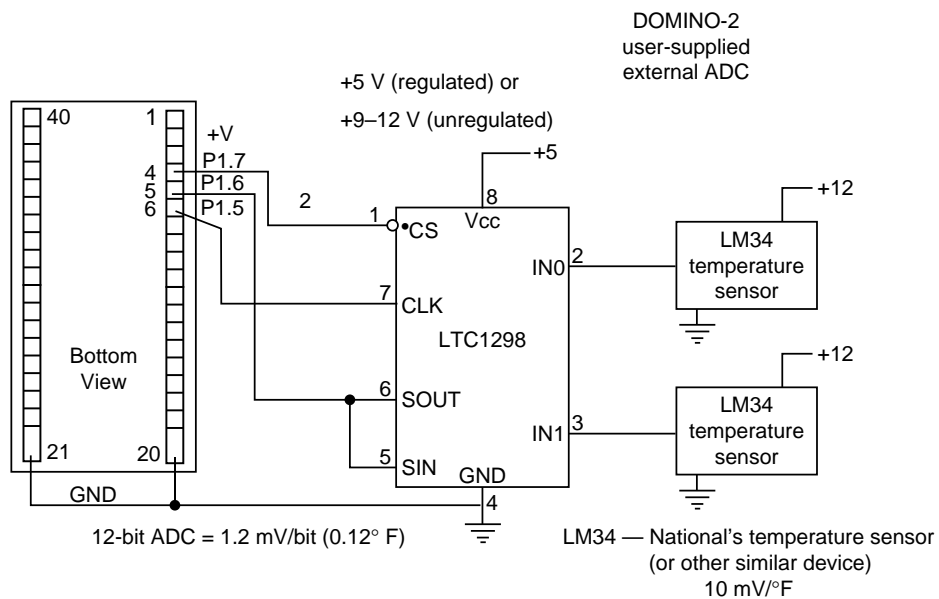
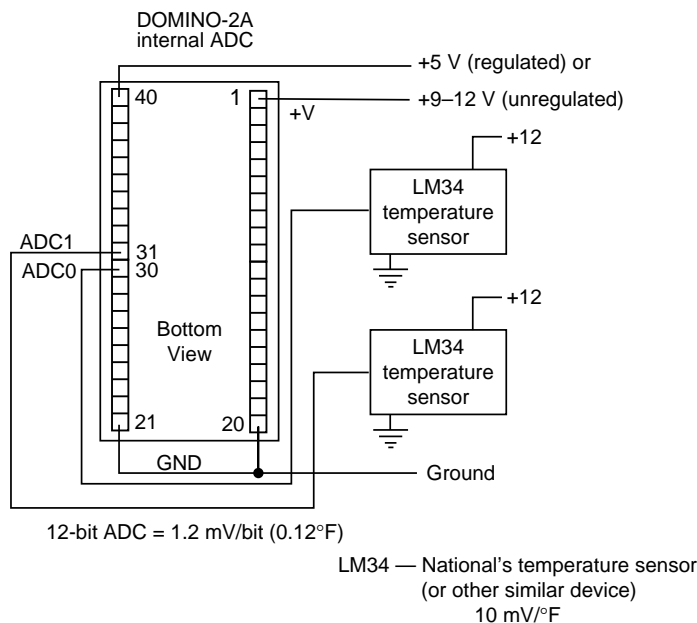


Figure 2—Typical connections for using DOMINO-2A with its internal ADC

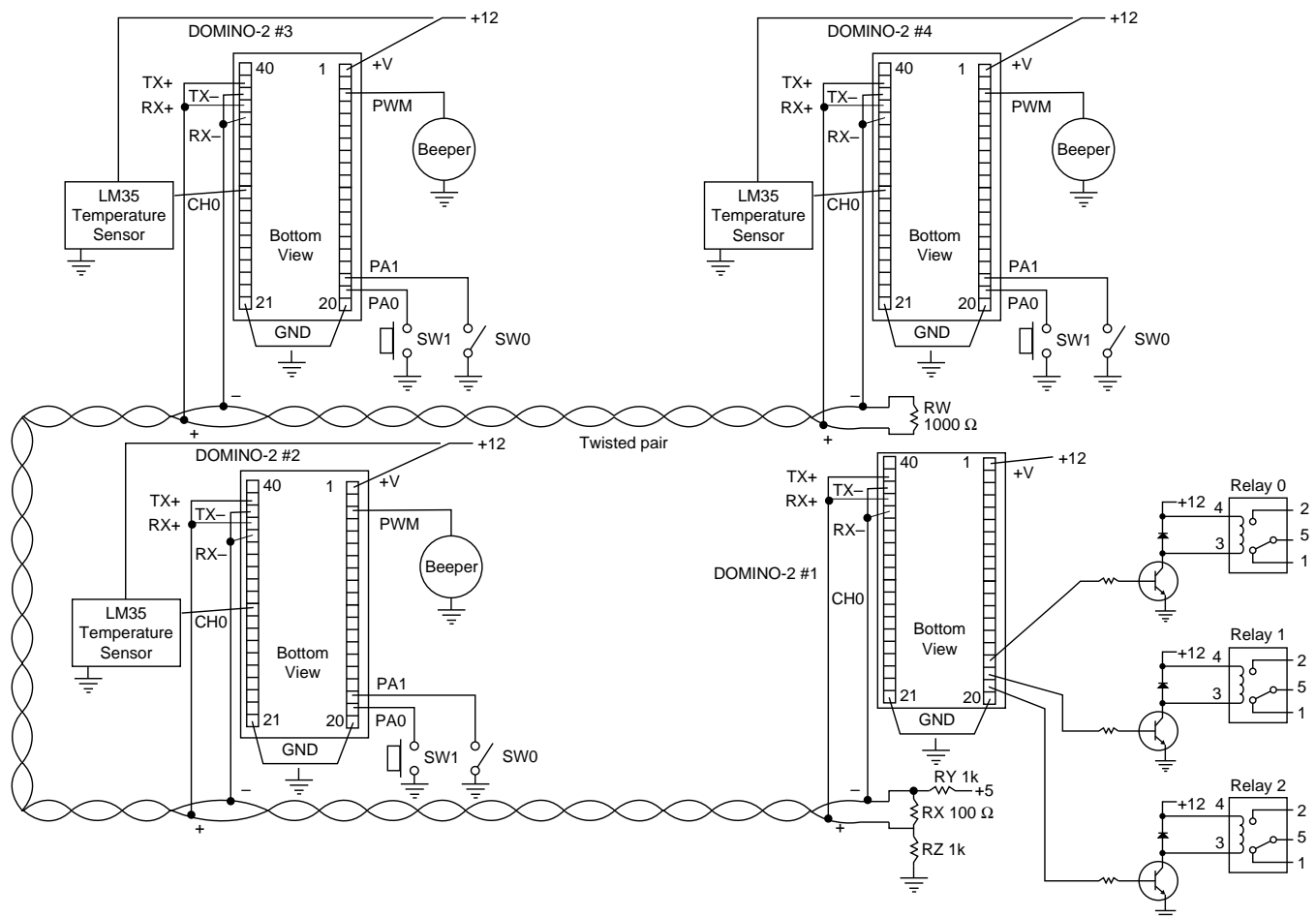


1.3 Sample Application: Networking DOMINO-2

Multiple DOMINO-2s can be used in a networked multidrop configuration using only a single twisted pair for communication. Network protocol requires that only one unit is allowed to transmit on the line at a time. All other units are listening in receive mode.

This is accomplished by requiring one DOMINO-2 or a device like a PC to be the net master. The master talks to any slave unit either passing information to it or requesting information from it. The slaves must never answer the mas-

ter until a response is requested. The master then relinquishes the net to that slave for the response and regains the net when the slave is finished. This arrangement enables multiple controllers to work together gathering numerous inputs and controlling innumerable outputs, independent of the system's size.



APPENDIX 2.0

Using Assembly Language Programs with DOMINO-2 (Utilities Version 1.3)

DOMINO-2 has become very popular in the world of embedded control. This is primarily due to the programming simplicity of DOMINO's onboard BASIC interpreter. DOMINO's user-friendly programming environment gets the application up and running in the shortest time possible. Finally, because the application can be stored as a nonvolatile auto-starting program directly in the onboard EEROM, virtually nothing is required to go from a development environment to turnkey production use.

For the most part we have succeeded in providing the proper combination of high-level language performance and ease of development and use. If users have ever expressed a wishlist of improvements, it has only been an interest in making it easier to incorporate and call assembly language programs along with BASIC.

From day one, of course, DOMINO has had the capability of calling an assembly language program. To get the program into memory requires entering the program via BASIC DATA statements and poking the program into a specific memory address. Such a procedure is acceptable for short programs.

As users apply DOMINO to increasingly sophisticated applications, the demand to go beyond BASIC and a few

small assembly routines has also increased. In Version 1.3 of the DOMINO utilities, we have tried to address the wishes of those who want to use larger assembly language programs. Of course, given the physical hardware, not everything is possible.

For a DOMINO-2 with V1.3 utilities (all DOMINOs can be upgraded to the latest utility version), the programming environment can be thought of as 4 different modes: BASIC only; BASIC with callable assembly routine(s) entered via DATA statements; BASIC with EEROM variable storage; and, BASIC with callable small or large assembly routine(s) entered via a hex download. The later 2 modes are new with version 1.3. The major difference afforded with V1.3 is that the entire contents of SRAM is now stored to EEROM, not just the BASIC program. If that SRAM contains a large assembly language program, both BASIC and that program will be saved. Please note that executing an assembly language program saved by that method requires the user to follow a few rules regarding the ORG addresses. More on this later. First, here is a description on each mode:

2.1 BASIC Only

DOMINO-2 was designed for use with the internally masked BASIC interpreter. For most applications the BASIC language will be sufficiently powerful.

2.2 BASIC with Included Routine in DATA Statements

Advanced programmers might wish to get down to a lower level to achieve maximum performance on a particular routine. A CALL from BASIC can be made to either an compiled or assembled routine. Small routines can be poked into protected SRAM using DATA statements. This approach keeps all the code, BASIC and assembly, in one easily maintainable file. The following illustrates an assembly program to toggle bit P3.5:

```
10 MTOP=1FFFFH : REM PROTECT MEMORY ABOVE
    1FFFFH
20 FOR X=2000H to 2005H : REM ADDRESSES TO
    PUT DATA
30 READ D : REM READ A BYTE OF DATA
40 XBY(X)=D : STORE THE DATA BYTE AT
    ADDRESS x
50 NEXT X : REM DO MORE
60 REM YOUR PROGRAM GOES HERE
.
```

```
100 CALL 2000H : REM THIS CALL SETS OUTPUT
    BIT P3.5 (T1)
.
.
200 CALL 2003H : REM THIS CALL CLEARS OUTPUT
    BIT P3.5 (T1)
.
.
999 END
1000 DATA 0D2H, 0B5H : REM SETB P3.5
1010 DATA 022H : REM RETURN
1020 DATA 0C2H, 0B5H : REM CLR P3.5
1030 DATA 022H : REM RETURN
```

2.3 BASIC and Large Assembly Code

At some point, your routines could become too large to include as DATA statements. The STOREHEX.BAS program provided will load your compiled or assembled .HEX file into SRAM for a debugging session if it was ORG'd for the SRAM addresses between 2000H and 6FFFH. Or, if it was ORG'd for the EEROM (between 9000H and 0DFFFH) STOREHEX.BAS will relocate the load properly into SRAM so it will be saved along with your BASIC program when using the BASIC program save command 'PROG' (utilities V1.3).

Placing your routine in SRAM for debugging:

Assemble the routine using an ORG=2000H (or higher but not extending beyond 6FFFH.)

Set MTOP to below ORG address to protect upper SRAM
MTOP=1FFFF (Remember to leave room for your BASIC code).

Use STOREHEX.BAS to load in yourfile.HEX for SRAM.

Use NEW command to delete STOREHEX.BAS

Load in your BASIC application.

Test application.

Note: BASIC program must fit below MTOP. Assembled routine must not extend beyond 6FFFH.

Placing your routine in SRAM for saving into EEROM:

Assemble the routine using an ORG=9000H (or higher but not extending beyond 0DFFFH.)

Set MTOP as previous to protect upper SRAM.

MTOP=1FFFF (Remember to leave room for your BASIC code).

Use STOREHEX.BAS to load in yourfile.HEX for EEROM.

Use NEW command to delete STOREHEX.BAS

Load in your BASIC application.

Note: BASIC program must fit below MTOP. Assembled routine must not extend beyond 0DFFFH. Save the BASIC program (and your routine) using one of the PROG calls (FF00h-FF20h).

2.4 Saving Configuration Variables

If your program needs to update some configuration information as part of the executing application, refer to SAVECFG.BAS program. You may include the appropriate parts of this program which allows up to 60 floating point numbers to be saved into 10 protected blocks of EEROM. 6 floating point numbers are stored into each 64 byte block using a storage routine loaded from DATA statements. The DATA statements are XBY'd into SRAM at 7000H and must execute from this location. A 64 byte block of SRAM is set aside at location 7FC0H-7FFFH. This PAGE is used as a transfer buffer for any variables stored by the BASIC application. A CALL 7000H unlocks and transfers this block of DATA into the EEROM. The user is responsible for placing the data into the block prior to calling the routine with the PAGE number (0-9) at which to store it within the EEROM. Although once stored the DATA can be retrieved (read) directly from the EEROM (0F800H-0F83FH for PAGE

0) you may wish to move it down to the 64 byte SRAM block to keep the bookkeeping straight.

Please keep in mind that unlike SRAM, EEROM can only be written to a finite number of times (like SRAM, however it can be read from an infinite amount). While the WRITE endurance of the EEROM is in excess of 1,000 times, a runaway routine could conceivably ruin the device in short order if allowed to WRITE in a loop. Therefore do not use the EEROM for DATA logging. If you must save something when the power is off, use the configuration storage variable technique described above just prior to powering the system down.

Each DATA statement in the BASIC program is one assembly language instruction. Line 1200 and 1210 actually set the 10 page beginning address. **Be careful if you make any changes, a simple error may cause permanent and irreversible damage to Domino, voiding any warranty.**

2.5 Files On This Diskette

UTIL_130.HEX INTEL.HEX file of the V.1.3 Utilities
(use LOAD_130.BAS to reload into DOMINO)

LOAD_130.BAS Program written in BASIC to reload the utilities (UTIL_130.HEX) into DOMINO

UTIL_130.DOC Revision summary and information on reloading the utilities into DOMINO

STOREHEX.BAS Program written in BASIC to load a compiled/ assembled routine into SRAM for debugging or temporarily into SRAM for storage to the EEROM
SAVECFG.BAS Program written in BASIC which demonstrates how floating point configuration values may be stored and updated to/from EEROM
README.TXT

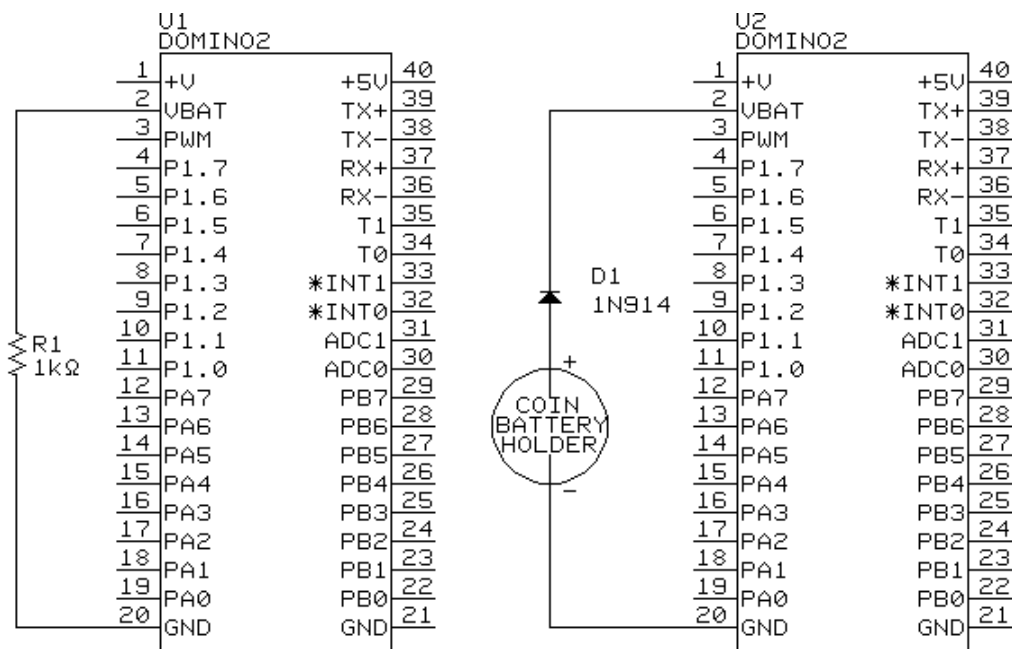
Errata

11/4/99

Domino 2 Modules

Starting from the date code 9944, the following engineering change has taken place on the Domino 2.

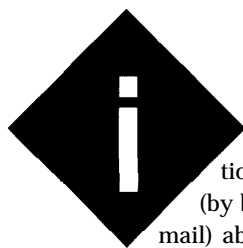
Occasionally the Domino 2 will experience difficulty in resetting the Co-Processor properly during a brief cycling of the power. To correct this problem Micromint has made a change to the Domino 2 that allows the user the option of a faster reset on the Co-Processor. U1 on the schematic demonstrates this. To battery back up the clock a 1N914 or equivalent diode must be placed externally on the Vbat line (U2 on the schematic). This only applies to Domino 2's with the date code of 9944 or later.



Intel Hex to BASIC Data Statement Translator

FROM THE BENCH

Jeff Bachiochi



get a ton of questions each month (by both phone and E-mail) about using masked

BASIC-52 on the 8052 microcontroller. The ever-increasing interest supports my claim that BASIC offers a familiar and friendly platform to learn embedded control. To the seasoned veteran, it also provides an inexpensive development platform.

The whole thing started back in 1984 when Intel masked an 8-KB control-oriented BASIC interpreter, called BASIC-52, into an NMOS 8052AH DIP-style microcontroller. While Intel no longer sells the chip, Micromint continues to offer BASIC-52 masked into low-power 80C52 DIP and PLCC packages.

with on-chip BASIC-52, writing applications is a snap. No special compilers or assemblers are needed. You just attach a terminal (or PC terminal emulator) and type the lines of BASIC in directly. The results can be stored and executed immediately right there on the target system.

Debugging the application is also painless since all variables can be displayed and BASIC lines edited at any time. For the majority of applications, BASIC is all you need to collect, transform, or redirect data.

Of course, no single programming language fits all control applications. What a BASIC interpreter brings in ease of use and program development, it compromises in execution speed and hardware to BASIC interfacing.

THE HARD FACTS

The 8031 core processor has four 8-bit I/O ports. In an 8052 processor

with the masked BASIC, Port0 and Port2 are used for the external address/data bus. All eight bits on Port1 are available through direct BASIC commands. The bits on Port3 have multiple functions and are available, but only through assembly instructions or assembly routines called from BASIC.

Many applications don't need more than eight I/O bits. However, if you need more, you can add external I/O peripheral chips. These can be easily accessed using traditional PEEK and POKE-type BASIC commands.

Some peripherals require interrupts for tasks which need to take precedence over the BASIC program flow. To facilitate this, BASIC-52 can directly respond to one of the two 8031 core external interrupts. It also can support a 1-s tick clock for interrupts based on elapsed time. The interrupt servicing speed remains that of BASIC.

THE NEED FOR SPEED

When the execution speed of a BASIC application program becomes time-critical, consider supplementing it with lower-level assembly language for speed-sensitive tasks. The typical execution time for a line of BASIC-52 is 230 ms, depending on the command. FOR/NEXT loops are the fastest while PRINT statements take considerably longer than the average.

Although assembly language executes in microseconds, it generally takes hundreds of lines of code to accomplish what a single line of BASIC can do.

On the other hand, task-specific assembly-language code (e.g., reading and storing A/D conversions) is much faster than interpreted BASIC (for a compiled BASIC the difference is not as significant).

CALL OF THE WILD

So, I contend that you should use a BASIC interpreter whenever and wherever it makes sense. When you need more execution speed, consider compiled BASIC or callable task-specific assembly language routines.

The BASIC-52 CALL 4200H statement saves a pointer to the next line of BASIC code on the stack and then jumps blindly to the address you give

it (in this case, 4200H). The processor now expects to fetch an assembly-language opcode to execute.

That's how your assembly routine gains control from BASIC. When your routine has finished, the **R ETURN** opcode returns control to BASIC. The pointer to the next line of BASIC is popped off the stack and execution of the BASIC application continues.

Let's assume that all you need to do is set and clear an I/O bit normally unavailable to BASIC, like T1 (P3.5). First, you need a place in memory to store the routine. You might want to place the routine in ROM above the space where the BASIC program resides in autostart mode.

There's one problem with this solution. You now have two programs which must be loaded properly, one BASIC and the other assembly language. While this may not sound like much of a problem, it can be a bookkeeping nightmare for longer programs, especially if you forget to keep the files together for easy maintenance.

I suggest an alternative approach. Try keeping the assembly routine as part of the BASIC application program using **DATA** statements. While this approach involves an extra step to protect the necessary RAM space and poke the routine into memory each time the application is run, the process is quite straightforward. Just look.

When you power up the 80C52 platform, you start out with an allocated address space like that in Figure 1a. The processor has measured the amount of RAM you have in the system and assigns it to the variable **MTOP** (let's assume **MTOP** = 7FFFH for a 32-KB SRAM).

Begin by typing in (or downloading) your BASIC-52 application. It fills

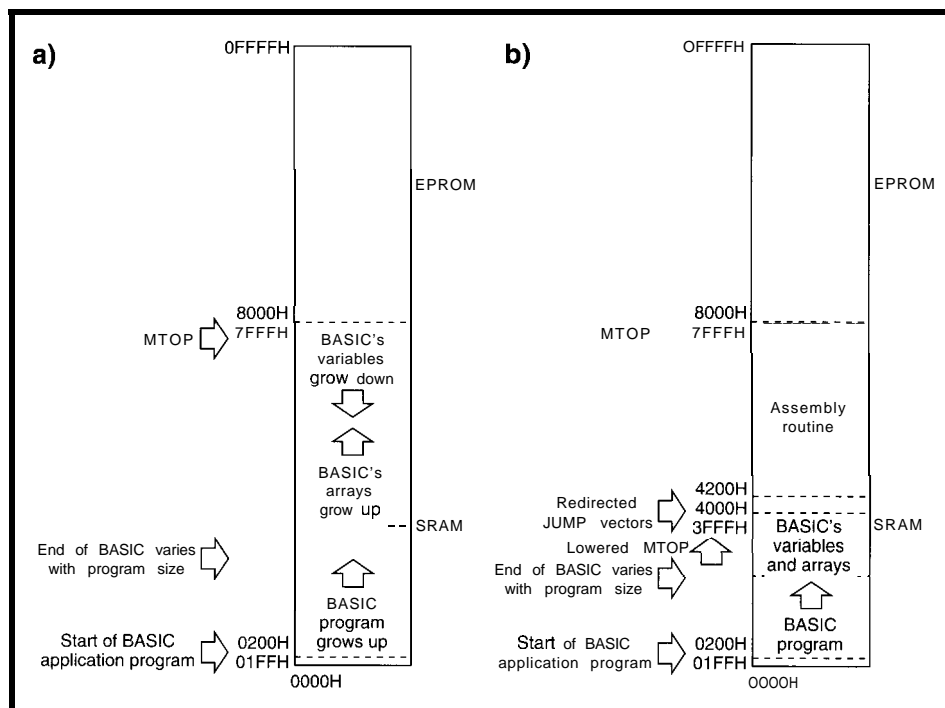


Figure 1-a) At powerup, BASIC puts variable storage as high in RAM as possible. **b)** Modifying **MTOP** protects a portion of memory for use by assembly language routines.

RAM from 200H upward. As the first statement, you need to add a line to protect some memory for the assembly-language routine.

This goal is accomplished by setting the **MTOP** variable to an address lower than that set in the power-up initialization. Let's use 3FFFH, to give you plenty of protected space.

10 MTOP=3FFFH

Notably, if your assembly-language routine were only three bytes long (and didn't require the use an interrupt), you would only have to protect three bytes (10 **MTOP**=7FFCH).

With **MTOP** reassigned to 3FFFH, you now have the address space allocated as in Figure 1b. Although you may not require the interrupt jump vectors which start at 4000H, I always protect them but leave them free of code. You may need them eventually. (More on this later.) To stay clear of

these locations, I started my code at 4200H.

Let's try something simple like turning on or off bit B5H (P3.5 T1), which you can't do directly from BASIC. You don't need an assembler for something this simple. It only requires two opcodes: a **SETB** (or **CLR**) instruction and a **RETURN**.

By referring to the micro's data book, you can find the correct bytes for setting and clearing a bit. You can place them into **DATA** statements like this:

```
10000 REM Set I/O bit T1
10010 DATA 0D2H, 0B5H:
      REM SETB T1
10020 DATA 022H: REM Return
10030 REM Clear I/O bit T1
10040 DATA 0C2H, 0B5H:
      REM CLR T1
10050 DATA 022H: REM Return
```

The first data byte, D2H, is the assembly-language opcode for setting an I/O bit. The second byte, B5H, is the bit address where the operation is to be performed. The source code for this

a) :20420000E4F590C2D5D2D47820C2D4759850758920758DFD858D8B75B81075A8 927588507A

b) : 20 4200 00 E4F590C2D5D2D47820C2D4759850758920758DFD858D8B75B81075A892758850 7A

Figure 2-a) A raw line of Intel hex looks like a jumble of characters. **b)** The line separated into its six major parts-start character, data length, load address, mode, data, and checksum-becomes easier to deal with.

opcode follows in a remark statement for documentation purposes only.

In the second statement, 22H is an opcode which returns control (in this case, to BASIC). This hand-coding method can be used when there is little chance for error.

You're welcome to hand-code larger routines, but be advised that it's extremely easy to miscode a statement, especially one with relative jumps and such. Do it as an exercise, and back it up with output from an assembler. It's bad enough when your routine doesn't run due to an error in logic. Don't add coding errors to your debugging session!

Now, all you need to do is get these six bytes into protected RAM where they'll be ready for you to call them. I've suggested using 4200H as the starting address. So, you need a BASIC-52 routine which pokes the data bytes into RAM at 4200H using the X BY statement. You can use a routine like this:

```
20 FOR X = 4200H TO 4205H
30 READ V
40 XBY(X) = V
50 NEXT X
```

The FOR/NEXT loop assigns 4200H to variable X. It reads a byte and places it into address location X. The address is incremented, and the read-and-store is repeated until X exceeds 4205H.

Once the data has been stored, it remains in RAM until something overwrites it or the power is cycled off and on. Your BASIC application can CALL 4200H to set T1 and CALL 4203H to clear T1.

You can even make the calls from the command-line prompt to test them. You quickly discover that if you make a call to a location which either has no routine or has a miscoded routine, anything can happen.

Anything can include totally locking up the system, so you may wish to both check your routine carefully and make sure it's there before you call it (at least the first time). At a minimum, at least ensure the first byte at the location you call is correct.

You can also sum all the code you placed in RAM and compare the total

Listing 1-This program, written in a generic PC BASIC, translates an Intel hex file into an 80C52 BASIC program and loads the Intel hex data into SRAM.

```
10 CLS
20 FLAG=0
30 REM This program prompts for an Intel hex file name,
40 REM reads the file in, and creates an output file. The
50 REM file can be appended to a 80C52 BASIC program to load your
60 REM assembly routine into SRAM (located in combined
70 REM Data/Code space) for execution there.
80 INPUT "What is the Intel hex filename? ",A$
90 OPEN A$ FOR INPUT AS #1
100 PRINT
110 PRINT "The output file will be called DATA.BAS."
120 INPUT "What line number should it begin with? ",LINENUMBER
130 B$ = "DATA.BAS"
140 OPEN B$ FOR OUTPUT AS #2
150 IF EOF(1) THEN GOTO 1: TEMP=LINENUMBER: GOTO 970
160 ON ERROR GOTO 950
170 INPUT #1, I$
180 IF (MID$(I$,1,1) <> ".") THEN GOTO 930
190 PAIRCOUNT = VAL("&H"+MID$(I$,2,2))
200 LOADADDRESS = VAL("&H"+MID$(I$,4,4))
210 GOSUB 610
220 MODE = VAL("&H"+MID$(I$,8,2))
230 IF (MODE<>0 AND MODE<>1) THEN PRINT "Warning, mode must be 00"
240 IF (MODE=1) THEN PRINT "End of File"
250 FOR X=10 TO 10+(2*(PAIRCOUNT-1)) STEP 2
260 IF (BYTECOUNT>7) THEN BYTECOUNT = 0: GOSUB 890:
    LINENUMBER = LINENUMBER+10
270 IF (BYTECOUNT=0) THEN G$ = "": TEMP=LINENUMBER: GOSUB 390:
    GOSUB 580
280 G$ = G$ + " 0" + MID$(I$,X,2) + "H"
290 TOTALSUM = TOTALSUM + VAL("&H"+MID$(I$,X,2))
300 IF (BYTECOUNT<>7) THEN G$ = G$ + ", "
310 BYTECOUNT = BYTECOUNT + 1
320 CHECKSUM = VAL("&H"+MID$(I$,10+2*(PAIRCOUNT-1),2))
330 FOR COUNT = 2 TO 10+(2*(PAIRCOUNT-1)) STEP 2
340 CHECKSUM = CHECKSUM + VAL("&H"+MID$(I$,COUNT,2))
350 NEXT COUNT
360 IF (CHECKSUM AND 255) <> 0 THEN PRINT "Checksum error"
370 NEXT X
380 GOTO 150
390 REM Place the line number digits into a string
400 BLANKFLAG = 0
410 IF (TEMP<10000) THEN GOTO 440
420 TEMPINTEGER = INT(TEMP/10000): G$ = G$ + CHR$(TEMPINTEGER+48)
430 TEMP = TEMP - TEMPINTEGER*10000: BLANKFLAG = 1
440 IF (TEMP<1000) THEN GOTO 470
450 TEMPINTEGER = INT(TEMP/1000): G$ = G$ + CHR$(TEMPINTEGER+48)
460 TEMP = TEMP - TEMPINTEGER*1000: BLANKFLAG = 1: GOTO 480
470 IF (BLANKFLAG=1) THEN G$ = G$ + "0"
480 IF (TEMP<100) THEN GOTO 510
490 TEMPINTEGER = INT(TEMP/100): G$ = G$ + CHR$(TEMPINTEGER+48)
500 TEMP = TEMP - TEMPINTEGER*100: BLANKFLAG = 1: GOTO 520
510 IF (BLANKFLAG=1) THEN G$ = G$ + "0"
520 IF (TEMP<10) THEN GOTO 550
530 TEMPINTEGER = INT(TEMP/10): G$ = G$ + CHR$(TEMPINTEGER+48)
540 TEMP = TEMP - TEMPINTEGER*10: BLANKFLAG = 1: GOTO 560
550 IF (BLANKFLAG=1) THEN G$ = G$ + "0"
560 G$ = G$ + CHR$(TEMP+48)
570 RETURN
580 REM Add the word "DATA" to the string
590 G$ = G$ + " DATA "
600 RETURN
610 REM Track the start and finish address for each segment
620 IF (FLAG<>0) THEN GOTO 650
630 START = LOADADDRESS: FINISH = LOADADDRESS + PAIRCOUNT 1:
    TOTALSUM
```

```

640 RETURN
650 IF (FINISH+1=LOADADDRESS) THEN FINISH = FINISH + PAIRCOUNT:
    RETURN
660 GOSUB 690
670 FLAG = 0
680 GOTO 610
690 REM Append the loading routine for the DATA statement segment
700 IF (RIGHT$(O$,1)=".") THEN O$ = MID$(O$,1,LEN(O$)-1)
710 IF (RIGHT$(O$,1)<>" ") THEN GOSUB 890:
    LINENUMBER = LINENUMBER + 10
720 O$ = " ": TEMP = LINENUMBER: GOSUB 390
730 O$ = O$ + " ST=0: CT=": TEMP = TOTALSUM GOSUB 390
740 GOSUB 890
750 LINENUMBER = LINENUMBER + 10
760 O$ = " ": TEMP = LINENUMBER: GOSUB 390
770 O$ = O$ + " FOR X = "
780 TEMP = START: GOSUB 390
790 O$ = O$ + " TO "
800 TEMP = FINISH: GOSUB 390
810 O$ = O$ + " : READ H : XBY(X)=H: ST=ST+H: NEXT X"
820 GOSUB 890
830 LINENUMBER = LINENUMBER + 10
840 O$ = " ": TEMP = LINENUMBER: GOSUB 390
850 O$ = O$ + " IF (CT<>ST) THEN PRINT "
860 O$ = O$ + CHR$(34) + "DATA ERROR" + CHR$(34) + ": END":
    GOSUB 890
870 LINENUMBER = LINENUMBER + 10: BYTECOUNT = 0
880 RETURN
890 REM Display and save present BASIC line
900 PRINT #2,O$
910 PRINT O$
920 RETURN
930 REM First character error in Intel hex paragraph
940 PRINT"Error. First character must be a ':'": CLOSE: END
950 REM Character error within Intel hex paragraph
960 PRINT"Error in input file": CLOSE: END
970 GOSUB 390: O$ = O$ + " RETURN": GOSUB 890: CLOSE: END

```

to a known good total placed in the BASIC program:

```

60 S = 0: C = 834
70 FOR X = 4200H TO 4205H
80 S = S +XBY(X)
90 NEXT X
100 IF (C<>S) THEN PRINT"Data
    error": STOP

```

This is where I lose a bunch of people. "I'm not gonna type in all those DATA statements with the code from my assembled source. My Intel hex file is over 1 KB in size."

There isn't much I can say that would convince them it would be worth their while. So, this month I present a piece of code, written in a generic PC BASIC, which reads in an Intel hex file and translates it into BASIC-52 DATA statements. The out-

put can be appended to your BASIC-52 application program.

INTEL HEX FILES

When a source file is assembled into a binary file, it contains no address information and no way-other than the file size-of assuring that the file has not been corrupted. When the binary file is translated into an Intel hex file, it becomes protected, if you will. The binary data is cut into small chunks, called *lines* or *paragraphs*, and surrounded by additional information.

As you can see in Figure 2, each Intel hex line begins with a ":" start character followed by the number of data bytes in the chunk (two hex characters) OOH-FFH. (Note that the chunk must have data bytes which can be loaded into successive addresses.) To

Table 2—80C52 BASIC remaps most of the interrupt vectors up to the 4000H area in code space.

Code space address	Function
4003H	IEO (external interrupt 0)
400BH	TFO (timer 0 overflow)
4013H	IE1 (external interrupt 1)
401 BH	TF1 (timer 1 overflow)
4023H	RI & TI (serial-port interrupt)
402BH	TF2 & EXF2 (timer 2/capture) (80 x 2 only)
4030H	UO1 (custom console output)
4033H	UI1 (custom console input)
4036H	(custom console status check)
403CH	PRINT@/LIST@
41 00H-41 FFH	CALL O-CALL 7FH

to do a health check on the data when it loads it into RAM from within my 80C52 application program.

Additional blocks are translated the same way. How, you might ask, can an Intel hex file contain more than one sequential block? For many files, it won't be. However, if you are using an interrupt [i.e., serial or timer), the processor has special preset locations it calls when an enabled interrupt occurs. Table 1 gives these interrupt-vector locations for the 8031 family.

In the 80C52, the masked BASIC has complete control of these locations (remember, code space 0000-1FFFH is internal). Knowing that users might want to access some of these interrupts, BASIC shadows the jump vectors to code space 4000H-41FFH, where they are in external address space. Since I use overlapping data and code space (*PSENOR'd with RD), this location is smack in the middle of RAM.

BASIC-52 application programs which don't use interrupts can safely have BASIC program lines up through this address space since the processor won't ever be calling this area. However, if you provide an assembly-language routine using the interrupts, you must protect the vector area by lowering MTOP to 3FFFH. The interrupt jump vectors are found at the locations shown in Table 2.

The vectors are called *interrupt jump vectors* because the user is expected to place an L J M P XXXX at the vector location to redirect the program flow to the beginning of their routine (i.e., XXXX).

For this reason, I like to stay out of the 4000H-41FFH area with my routines. Also, this use of vectors explains how assembled code can be nonsequential. If your assembly routine was

placed right at the jump vector location, it would

Take a look at Figure 3 to see how this problem is handled in the D E M O 1. H E X file translation. Figure 3 shows what happens when you run it. The DATA. BAS file that is created can now be appended to your application. Remember to add the M T O P statement to your application to protect some RAM and add a G O S U B 10000 statement so this appended portion loads the routines into protected RAM.

Here's a quick overview of just what the translation program does. Skip over lines 10000-10030 for a moment. Notice lines 10040-10070. Three data bytes are loaded into 0023H-0025H, an L J M P 424DH. This is the serial-port interrupt. To allow BASIC to connect, you must change the load values from X=35 T O 37 (23H-25H) to X=16419 T O 16421 (4023H-4025H).

Next, a second jump vector is loaded (lines 10080-10110) at 000BH-000DH. This L J M P 4242H is the timer-0 overflow interrupt. Change line 10100 from X=11 T O 13 (000BH-000DH) to X=16395 T O 16397 (400BH-400DH) to allow BASIC the hooks for this routine.

The main body of data consists of the actual routines which reside at 4200H (line 290, X=16896). No changes need to be made here. In fact, if your routines do not use interrupts, this one block of code is all you most likely will see.

Now, back to the first few lines. This code was written to be in total control of the processor and, as such, would normally get control from the reset vector at 0000H. The first chunk

of code is the reset vector jump. Since BASIC-52 runs on powerup and this vector is not shadowed like the others, it can be discarded. Well, almost.

Instead, 4200H is the location BASIC would call to enter the routine. In this example, there is never a return to BASIC. The assembly-language routine completely takes over until power is shut down. Here, BASIC loads the jump vectors and routines-once it passes execution over to the routine, it is never heard from again.

This situation, of course, is the extreme. Why bother at all with BASIC once you are writing totally in another language?

And rightly so. I do not advocate the use of BASIC for every application. But, I do like the friendly development environment and the ease of getting an application up and running.

As you have seen here, it is very possible for BASIC-52 and assembly routines to coexist. I hope I have demonstrated a way you can use the 80C52 to have your cake (the power of BASIC) and eat it too (call on the speed of assembly language). Remember, when all you need to do is tap, don't use a sledge. □

Jeff Bachiochi (pronounced "BAH-key-AH-key") is an electrical engineer on Circuit Cellar INK's engineering staff. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circellar.com.

IRS

428 Very Useful
429 Moderately Useful
430 Not Useful

What is the Intel hex filename? DEM01.HEX

The output file will be called DATA.BAS. What line number should it begin with? 10000

```
10000 DATA 002H,042H,000H
10010 ST=0: CT=68
10020 FOR X = 0 TO 2 : READ H : XBY(X)=H: ST=ST+H: NEXT X
10030 IF (CT<>ST) THEN PRINT "DATA ERROR": END
10040 DATA 002H,042H,04DH
10050 ST=0: CT=145
10060 FOR X = 35 TO 37 : READ H : XBY(X)=H : ST= ST+H: NEXT X
10070 IF (CT<>ST) THEN PRINT "DATA ERROR" : END
10080 DATA 002H,042H,042H
10090 ST=0: CH=134
10100 FOR X = 11 TO 13 : READ H : XBY(X)=H: ST= ST+H: NEXT X
10110 IF (CH<>ST) THEN PRINT "DATA ERROR": END
10120 DATA 0E4H,0F5H,090H,0C2H,0D5H,0D2H,0D4H,078H
10130 DATA 020H,0C2H,0D4H,075H,098H,050H,075H,089H
10140 DATA 020H,075H,08DH,0FDH,085H,08DH,08BH,075H
10150 DATA 0B8H,010H,075H,0A8H,092H,075H,088H,050H
10160 DATA 012H,042H,03FH,0FAH,0FBH,0F5H,090H,0F4H
10170 DATA 0F5H,08CH,0EBH,020H,0D5H,004H,003H,002H
10180 DATA 042H,033H,023H,0FBH,0F5H,090H,012H,042H
10190 DATA 03FH,06AH,060H,0EEH,06AH,080H,0E4H,0E5H
10200 DATA 0A0H,022H,0COH,0EOH,0B2H,0D5H,0EAH,0F4H
10210 DATA 0F5H,08CH,0DOH,0EOH,032H,0COH,0EOH,0COH
10220 DATA 0D0H,0C2H,098H,075H,0DOH,010H,0E5H,099H
10230 DATA 0F5H,0F0H,064H,0ODH,070H,0OFH,074H,01FH
10240 DATA 0C3H,098H,0F4H,060H,013H,0FBH,0E4H,018H
10250 DATA 0F2H,0DBH,0FCH,080H,0OBH,074H,03FH,0C3H
10260 DATA 098H,0B3H,050H,004H,0E5H,0F0H,0F2H,008H
10270 DATA 0D0H,0D0H,0D0H,0EOH,032H
10280 ST=0: CT=18439
10290 FOR X = 16896 to 17020 : READ H : XBY(X)=H : ST=ST+H :
NEXT X
10300 IF (CHECKTOTAL<>STORETOTAL) THEN PRINT "DATA ERROR" : END

End of file OK
```

Figure 3-A run of the translation program demonstrates the kind of output you can expect from an Intel hex file.