CTM-05
User's Guide

CTM-05
User's Guide

Revision D - December 1993
Part Number: 71930

The information contained in this manual is believed to be accurate and reliable. However, the manufacturer assumes no responsibility for its use; nor for any infringements or patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of the manufacturer.

THE MANUFACTURER SHALL NOT BE LIABLE FOR ANY SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RELATED TO THE USE OF THIS PRODUCT. THIS PRODUCT IS NOT DESIGNED WITH COMPONENTS OF A LEVEL OF RELIABILITY THAT IS SUITED FOR USE IN LIFE SUPPORT OR CRITICAL APPLICATIONS.

Contents

■ ■ ■

Chapter 1

INTRODUCTION

1.1 GENERAL

The CTM-05 is a multi-function counter-timer and digital expansion board for the IBM PC and compatibles. The board offers five 16-bit up/down counters, a 1MHz crystal timebase with divider, and separate general-purpose 8-bit TTL input/output (I/O) ports. An Advanced Micro Devices AMD-9513 System Timing Controller IC (Integrated Circuit) supports the board's counting and timing functions.

1.2 FEATURES

- Five independent 16-bit up/down counters.
- 7 MHz maximum input frequency.
- Binary or BCD counting.
- 1 MHz internal crystal oscillator with tapped scaler.
- Programmable frequency output.
- Time-Of-Day option.
- Alarm comparators on Counters 1 and 2.
- Complex-duty-cycle outposts.
- One-shot or continuous outputs.
- Programmable count gate/source selection.
- Programmable input and output polarities.
- Programmable gate functions.

Additional functions separate from the counter-timer include the following:

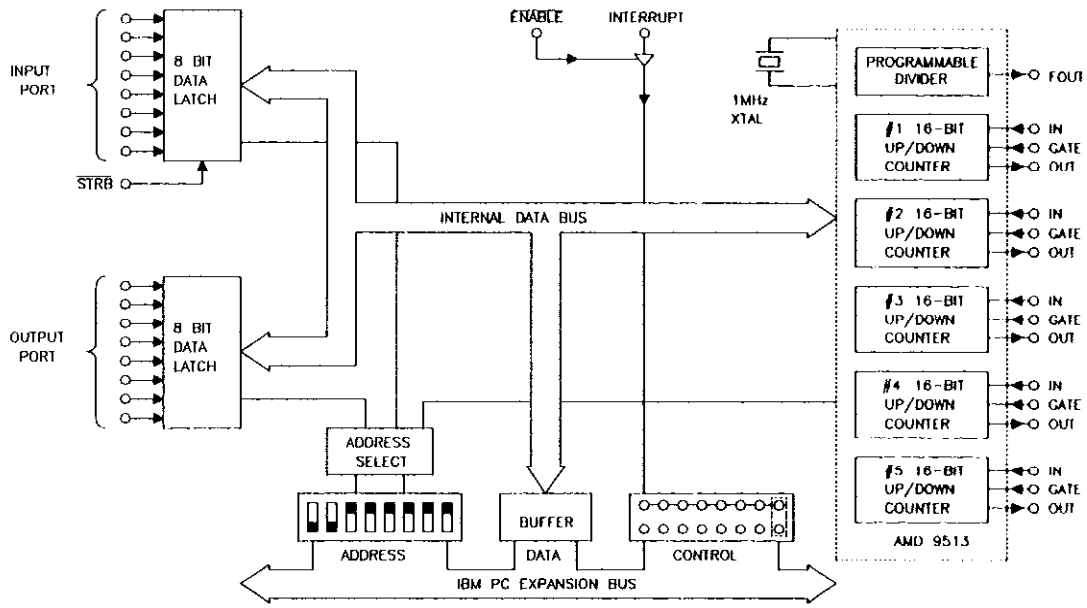
- Eight bits of TTL/DTL digital input with latch.
- Eight bits of TTL/DTL digital output with latch.
- Level-selectable interrupt input channel.

1.3 APPLICATIONS

- Event counting for pulse-output devices (flowmeters, wattmeters, etc.).
- Programmed frequency synthesis.

- Coincidence alarms.
- Frequency measurements.
- F/V conversion and pulse accumulation.
- Period and pulse duration measurements.
- Time delay generation.
- Periodic interrupt generation.
- Frequency Shift Keying (FSK).

1.4 BLOCK DIAGRAM



1.5 OPTIONAL I/O ACCESSORIES

To simplify complex I/O connections, an optional screw-connector board (STA-U) connects to the CTM-05 Board I/O connector via a flat-insulation displacement cable (C-1800).

1.6 SPECIFICATIONS

Counter Timer AM9513 (Advanced Micro Devices)

Five counter/timers:
- Independent or Cascadable.
- Programmable as up or down counters in either Binary or Binary Coded Decimal (BCD).
- Programmable to count on positive or negative edge.
- Programmable output polarity.
- Programmable gating on either logic level or edge.
- Selectable counter input clock.

Time Base	1.00MHz (±0.01% from 0 to 70°C.).
External Inputs	7.00MHz Max TTL.
Digital I/O	Latched LSTTL.
Ext. Int. & Enable	TTL.
Power Consumption	5.0V @ 450mA typical.
Size & Weight	5" (12.7cm) by 4.25" (10.80cm).
Environmental	0 to 70°C, 0 to 90% Humidity (Non-condensing).

■ ■ ■

Chapter 2

INSTALLATION

2.1 GENERAL

CTM-05 distribution software is on 5.25", 360K floppy diskette(s) and on a 3.5" diskette (DOS 2.10 format). This software is licensed to permit multiple copies for non-commercial use, not for resale.

Installation of your CTM-05 Software will require the following procedures:

- Making a working copy of your CTM-05 Distribution diskette(s).
- Unpacking and inspecting the board.
- Selecting a Base Address for your CTM-05 driver board.
- Installation.

2.2 COPYING THE DISTRIBUTION SOFTWARE

As soon as possible, make a back-up copy of your Distribution Software. With one (or more, as needed) formatted diskettes on hand, place your Distribution Software diskette in your PC's A Drive and log to that drive by typing `A:` . Then, make your backup using the DOS *COPY* or *DISKCOPY* command, as described in your DOS reference manual (DISKCOPY is preferred because it copies diskette identification, too).

2.3 INSPECTING

After you remove the wrapped board from its outer shipping carton, proceed as follows:

1. Place one hand firmly on a metal portion of the computer chassis (the computer must be turned Off and grounded). You place your hand on the chassis to drain off static electricity from the package and your body, thereby preventing damage to board components.
2. Allow a moment for static electricity discharge; carefully unwrap the board from its anti-static wrapping material.
3. Inspect the board for signs of damage. If any damage is apparent, return the board to the factory.
4. Check the contents of your CTM-05 package against its packing list to be sure the order is complete. Report any missing items to the manufacturer immediately.

2.4 SELECTING AND SETTING THE BASE ADDRESS

The CTM-05 requires four consecutive address locations in I/O space. Since some I/O address locations are already occupied by internal I/O and other peripheral cards, you have the option of resetting the CTM-05 I/O base address by means of an on-board Base Address DIP switch. The Base Address switch is located as shown in Figure 2-1, and it appears as shown in Figure 2-2. Referring to Figure 2-2, you set the base address on a four-byte boundary to 3FC Hex (300 Hex is shown).

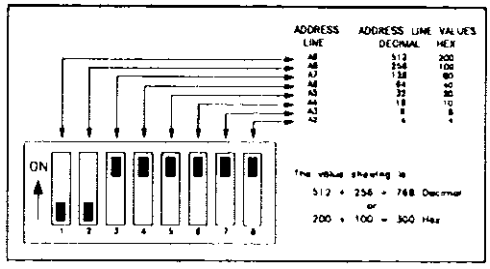


Figure 2-2. Base Address switch.

The board is preset for a base address of 300 HEX. If this address is not satisfactory, your distribution software contains a program called called *DIPSW.EXE* that asks for base address and shows a picture of the DIP switch setting. Use this program by logging to its location (to the floppy drive containing the distribution diskette or to the hard-drive directory containing your distribution files) and typing **DIPSW**

When the computer responds with *Desired base address?*, type your choice in decimal or IBM &H___ format and press < *Enter* > .

2.5 SETTING THE INTERRUPT LEVEL

The Interrupt Jumper Block is above the board's edge connector. If your application does not require interrupts, place the Interrupt Jumper in the rightmost position of the Jumper Block (Position X in the block diagram of Chapter 1; X = inactive position). If you wish to use interrupts, select the level from Positions 2-7 of the Jumper Block.

Avoid using a level already in use by another device (for example, Level 6 is always used by the floppy-disk drives), unless you are using programming that allows several devices to share one level.

The tri-state driver is enabled by taking the Interrupt Enable (Pin 2) to a logic low level. A positive edge on the Interrupt input (Pin 1) will then generate an interrupt after the 8259 Interrupt Controller is enabled.

The use of interrupts implies that the user has installed an Interrupt Service Routine and Interrupt Vectors to the Service Routine. It also implies that the user has enabled the 8259 Mask Register for the selected level. You must use Assembly Language to set up an Interrupt Service Routine, as it is impossible to program this function in the BASIC Language.

2.6 HARDWARE INSTALLATION

To install the CTM-05 in a PC, proceed as follows.

> **WARNING:** ANY ATTEMPT TO INSERT OR REMOVE ANY ADAPTER BOARD WITH THE COMPUTER POWER ON COULD DAMAGE YOUR COMPUTER!

1. Turn Off power to the PC and all attached equipment.
2. Remove the cover of the PC as follows: First remove the cover-mounting screws from the rear panel of the computer. Then, slide the cover of the computer about 3/4 of the way forward. Finally, tilt the cover upwards and remove.
3. Choose an available option slot. Loosen and remove the screw at the top of the blank adapter plate. Then slide the plate up and out to remove.
4. Hold the CTM-05 board in one hand placing your other hand on any metallic part of the PC chassis (but not on any components). This will safely discharge any static electricity from your body.
5. Make sure the board switches have been properly set (refer to the preceding section).
5. Align the board connector with the desired accessory slot and with the corresponding rear-panel slot. Gently press the board downward into the socket. Secure the board in place by inserting the rear-panel adapter-plate screw.
7. Replace the computer's cover. Tilt the cover up and slide it onto the system's base, making sure the front of the cover is under the rail along the front of the frame. Replace the mounting screws.
8. Plug in all cords and cables. Turn the power to the computer back on.

2.7 I/O CONNECTOR PIN ASSIGNMENTS

Pin assignments of the 37-pin, D, male connector are as follows:

PIN NO.	SIGNAL	PIN NO.	SIGNAL
19	Source 2	37	Counter 1 Gate
18	Counter 2 Gate	36	Source 1
17	Source 3	35	Counter 1 Output
16	Counter 3 Gate	34	Counter 2 Output
15	Source 4	33	Counter 3 Output
14	Counter 4 Gate	32	Counter 4 Output
13	Source 5	31	Counter 5 Output
12	Counter 5 Gate	30	FOUT
11	Digital Common	29	IP0
10	OP0	28	IP1
9	OP1	27	IP2
8	OP2	26	IP3
7	OP3	25	IP4
6	OP4	24	IP5
5	OP5	23	IP6
4	OP6	22	IP7
3	OP7	21	Input Strobe
2	/Interrupt Enable	20	+5V Power (from PC)
1	Interrupt Input		

The mating connector must be a 37-pin, D, female. Specifically, the connector must be a solder-cup type ITT/Cannon DC-37S (or #SFC-37 from the manufacturer). Cabling must be flat-cable type Amp #745242-1.

■ ■ ■

Chapter 3

REGISTER FUNCTIONS & LOCATIONS

3.1 I/O MAP

The CTM-05 uses four consecutive addresses in the PC's I/O address space. The base (or starting) address is set by the Base Address Switch (see Chapter 2) and automatically falls on a 4-bit boundary. Once the base address is set, the four consecutive addresses are used as follows:

I/O ADDRESS		FUNCTION WRITE	READ
Base	+0	9513 Data In	9513 Data Out
	+1	9513 Command Reg.	9513 Status Reg.
	+2	- - -	IP0-7 Digital Input
	+3	OP0-7 Digital Output.	

IBM PC-AT users should note that all ports are 8-bit (one byte) and should perform byte-oriented read/write operations rather than word (16-bit) operations. When performing consecutive byte transfers to the same PC-AT I/O port (common with 9513 architecture), the *IBM PC-AT Technical Reference Manual* recommends the following Assembly Language statements to allow sufficient recovery time for the AT I/O circuits.

```
        OUT IO_ADDR,AL      'WRITE LOW BYTE
        JMP NEXT            'DELAY
NEXT:   MOV AL,AH           'FETCH HIGH BYTE
        OUT IO_ADDR,AL      'WRITE HIGH BYTE
```

3.2 INTRODUCTION TO THE 9513

This section provides general information on how the 9513 counter/timer is used on the CTM-05 and PCF-CTM-05. For detailed information on programming the 9513, refer to the AM9513 technical manual, available from the following source: Advanced Micro Devices, 901 Thompson Place, PO Box 3453, Sunnyvale, CA 94088, 800/538-8450. All data transfers to the 9513 timer-counter use two I/O ports. Data transfer uses the port at the Base address; for example, loading and reading counters and counter mode registers. The port at Base address +1 carries addressing, command, control, and status. The many internal registers of the 9513 require an indirect system of access using a Data Pointer Register, which is reached via the Command Register. The Command Register also performs other functions such as loading and enabling the counters, latching counter contents, etc. Acceptable Command Register codes are listed in the following table.

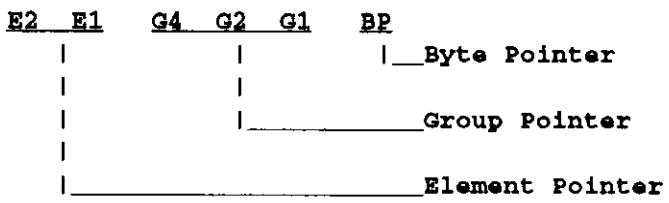
COMMAND CODE								
C7	C6	C5	C4	C3	C2	C1	C0	FUNCTION
0	0	0	E2	E1	G4	G2	G1	Load Data Pointer Register with E % G.
0	0	1	S5	S4	S3	S2	S1	Arm counting for selected counters (S = 1).
0	1	0	S5	S4	S3	S2	S1	Load source into specified counter.
0	1	1	S5	S4	S3	S2	S1	Load and arm specified counters.
1	0	0	S5	S4	S3	S2	S1	Disarm and save all selected counters.
1	0	1	S5	S4	S3	S2	S1	Save selected counters in hold registers.
1	1	0	S5	S4	S3	S2	S1	Disarm all selected counters.
1	1	1	0	0	N4	N2	N1	Clear Output Bit (001 <= N <= 101).
1	1	1	0	1	N4	N2	N1	Set Output Bit N (001 <= N <= 101).
1	1	1	1	0	N4	N2	N1	Step Counter N (001 <= N <= 101).
1	1	1	0	0	0	0	0	Enable Data Pointer sequencing (clear MM14).
1	1	1	0	0	1	1	0	Gate FOUT on (clear MM12).
1	1	1	0	0	1	1	1	Enter 8-bit bus mode (clear MM13).
1	1	1	0	1	0	0	0	Disable Data Pointer sequencing (set MM14).
1	1	1	0	1	1	1	0	Gate FOUT off (set MM12).
1	1	1	0	1	1	1	1	Enter 16-bit bus mode (set MM13).
1	1	1	1	1	1	1	1	Master reset.

Note the following logical structure in the command codes:

- All codes beginning with 000: Reference Data Pointer Register.
- Codes from 001 to 110: Reference counter operations.
- Codes beginning with 111 and ending with 001-101: Perform single-bit counter functions
- Codes beginning with 111 and ending with 000 or 110-111: Perform master control functions (all these functions can also be activated by writing the Master Mode Register).

Those codes that reference counter operations use a linear select S5-S1. Only the counters with the appropriate S bit set are affected. This is a powerful feature in that it allows simultaneous loading, latching, enabling, etc. of any combination of the 9513 internal counters.

Returning to command codes that commence with 000. These codes select the internal registers according to E and G fields that set the Internal Data Pointer Register. The 9513 has one Master Mode Register that controls the operation of all counters and the scaler. This must be set in the initialization sequence of your program. In addition, each counter has its own mode, load, and hold registers. These registers are accessed through the data port at the Base address after setting the Internal Data Pointer Register tp address the desired register. The data format of the Data Pointer Register is as follows:



These pointers are defined as follows:

Byte Pointer	1 = Least significant byte transferred next. 0 = Most significant byte transferred next.
Group Pointer	G4, G2, and G1: 000 - Illegal 001 - Counter Group 1 010 - Counter Group 2 011 - Counter Group 3 100 - Counter Group 4 101 - Counter Group 5 110 - Illegal G4, G2, and G1: 111 - Always for control group.
Element Pointer	Counter Group E2 and E1: 00 - Mode Register 01 - Load Register 10 - Hold Register 11 - Hold Register/Hold Cycle Increment Control Group E2 and E1: 00 - Alarm Register 1 01 - Alarm Register 2 10 - Master Mode Register 11 - Status Register/No Increment

The Data Pointer consists of a 2-bit Element Pointer (E), a 3-bit Group Pointer (G), and a 1-bit Byte Pointer (B). The Byte Pointer bit indicates which byte of a 16-bit register is to be transferred on the next access through the data port. Whenever the Data Pointer is loaded, the Byte Pointer (B) is set to 1, indicating a least significant byte of data is expected next. With an 8-bit data bus (as used on the IBM PC), the Byte Pointer toggles after each 8-bit data transfer Master Mode Bit MM13 = 0). The Element and Group Pointers together select the internal register that is to be accessible through the data port. Although the Element and Group Pointers in the Data Pointer Register cannot be read, the Byte Pointer is available as a bit in the Status Register.

Random access to any internal location can be achieved by loading the Data Pointer (through Base address +1) and then reading or writing to the location through the data port (at Base address) as appropriate. The Counter Registers are all 16-bit and after loading the pointer, data is transferred in low-byte/high-byte sequence. The following example shows loading Counter 3 Load Register (using BASIC):

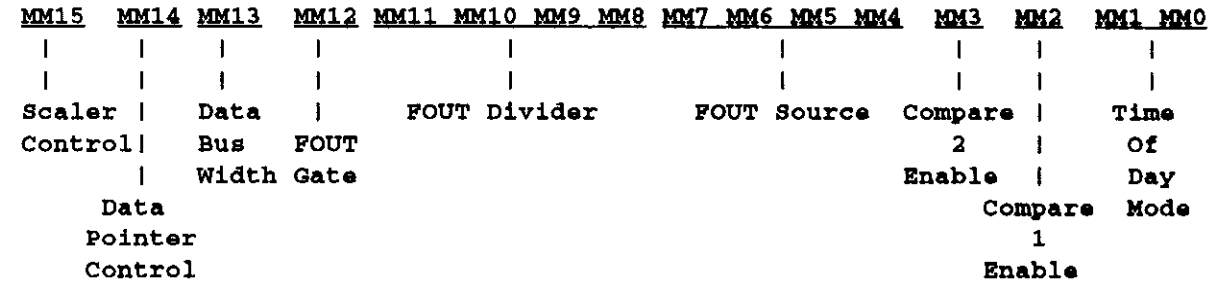
```
xxx10   OUT BASE + 1, &H13    'write 000 10 011 to command reg.
xxx20   OUT BASE, 0           'low byte = 0
xxx30   OUT BASE, &H80        'high byte = 128 register loaded
                                    'with 32,768
```

Many programs contain a pattern of loading the Counter Mode Register, the Load Register, and the Hold Register in sequence or setting Alarm Register 1, Alarm Register 2, and the Master Mode Register. The Element Pointers are arranged to auto-increment on each 2-byte data transfer if Master Mode Bit 14 (MM14) = 0. This saves writing to the Command Register between items of data and, depending on your preferences, is a feature that you may wish to use for brevity of code or to ignore for clarity of code.

In general, most programs will consist of an initialization section that sets the overall operation of the 9513 through the Master Mode Register, then sets each counter operating configuration through its individual mode register, and finally loads initial data into the counters through the Load or Hold Registers. Following the initialization, the counters are usually enabled using the Command Register, possibly latched and read using the Command and Hold Registers, etc. or disabled, re-loaded, and re-enabled, etc. Most "heavy" work in programming is in the initialization; subsequent reading and writing operations are much simpler. An example of CTM-05 set up as a straight, 5-channel, up counter is in program COUNT2.BAS.

3.3 MASTER MODE REGISTER

The Master Mode Register controls the overall operation of the 9513 and should be the first register initialized by your program. The register is 16-bits that function as follows:



These bits function as follows:

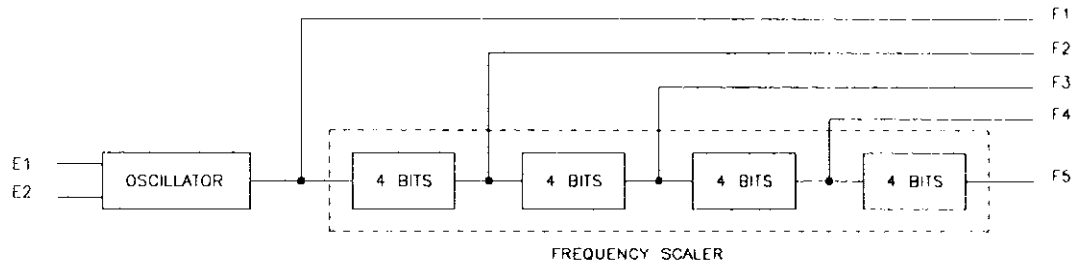
Scaler Control	0 = Binary Division 1 = BCD Division
Data Pointer Control	0 = Enable Increment 1 = Disable Increment
Data Bus Width	0 = 8-Bit Data Bus 1 = 16-Bit Data Bus
FOUT Gate	0 = FOUT On 1 = FOUT Off (Low to Ground)

FOUT Divider	0000 - Divide By 16 0001 - Divide By 1 0010 - Divide By 2 0011 - Divide By 3 0100 - Divide By 4 0101 - Divide By 5 0110 - Divide By 6 0111 - Divide By 7 1000 - Divide By 8 1001 - Divide By 9 1010 - Divide By 10 1011 - Divide By 11 1100 - Divide By 12 1101 - Divide By 13 1110 - Divide By 14 1111 - Divide By 15
FOUT Source	0000 - F1 0001 - Source 1 0010 - Source 2 0011 - Source 3 0100 - Source 4 0101 - Source 5 0110 - Gate 1 0111 - Gate 2 1000 - Gate 3 1001 - Gate 4 1010 - Gate 5 1011 - F1 1100 - F2 1101 - F3 1110 - F4 1111 - F5
Compare 2 Enable	0 = Disabled 1 = Enabled
Compare 1 Enable	0 = Disabled 1 = Enabled
Time Of Day Mode	00 = TOD Disabled 01 = TOD Enabled /5 Input 10 = TOD Enabled /6 Input 11 = TOD Enabled /10 Input

MM15 selects the dividers for the four counters in the Crystal Oscillator Scaler. The Scaler stages can divide by either 10 or 16 (BCD or binary) according to whether MM15 is 1 or 0. The fundamental crystal frequency F1 (1MHz) and each of the scaler outputs F2, F3, F4, and F5 can be routed to any of the counters and the FOUT divider by software control. For instance with MM15 = 1 (BCD), the frequencies will be:

F1	F2	F3	F4	F5
1MHz	100KHz	10KHz	1Khz	100Hz

The structure of the Oscillator Scaler is shown below.



MM14 selects automatic incrementing of the Data Pointer Register. MM14 can also be individually controlled via the Command Register.

MM13 selects the Data Bus Width and for IBM PC operation should always be zero (8-bit bus). MM13 can also be individually controlled by the Command Register.

MM12 controls operation of FOUT (Pin 30 on the CTM-05). When MM12 is low, FOUT is enabled. When MM12 is high, FOUT is at a logic low (note this is not a tristate output). MM12 can also be individually controlled via the Command Register.

MM11 through MM8 set the divider modulus for the FOUT divider (not to be confused with the oscillator scaler). This is a 4-bit divider counter ahead of the FOUT output. Any modulus from 1 to 1 is possible.

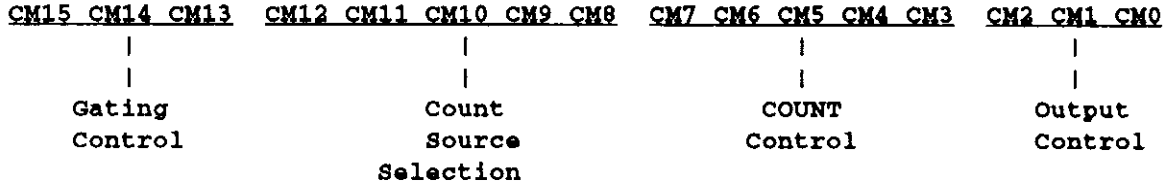
MM7 through MM4 set the input source of the FOUT divider. This can be any of the Oscillator Scaler outputs F1-F5, any of the Counter Gate inputs GATE 1 - 5, or any of the external source inputs SOURCE 1 - 5. Truly flexible!

MM3 and MM2 set the comparison modes for Counters 2 and 1. If these bits are set, the comparator outputs are substituted for the normal counter outputs on Counter Out 1 and 2 (Pins 35 and 34). The comparator output will be active high if the Output Control Field of the Counter Mode Register is 001 or 010 and active low for a code of 101. Once the Compare output is true, it will remain so until the count changes and the comparison therefore goes false.

Finally, MM1 and MM0 set the optional Time Of Day Mode for Counters 1 and 2. When both these bits are zero, Counters 1 and 2 operate in exactly the same way as all the other counters. For other combinations of these bits, the counter division ratios are set so that the most significant byte of Counter 2 is hours, the less significant byte is minutes and the most significant byte of Counter 1 is seconds. The least significant byte section of Counter 1 becomes a pre-scaler in this mode and can divide by 50, 60, or 100 for 50Hz, 60Hz, or 100Hz (crystal) input frequencies.

3.4 COUNTER MODE REGISTERS

Each counter has its own mode-register controls to control its operation. The Counter Mode Registers should be initialized after the Master Mode Register. Each register is 16 bits, as follows.



CM13-15 control the effect of the GATE inputs on the selected counter. The gate input can be disabled (000) or enabled in a variety of ways. The counter can be gated for counting from the previous counter (TCN-1 = Terminal Count of Counter - 1); for example, Counter 3 could be gated by the output of Counter 2. Alternatively, the counter can be gated from its own gate input (GATE N) or adjacent gate inputs (GATE N-1 or N+1). This last configuration allows 2 or 3 adjacent counters to share the same gate control input provided the gate is level triggered. If only the counter's own gate input is used, it may be level-triggered (active high or low) or edge-triggered (positive or negative).

CM8-12 control the clock input source for the counter. You can select whether you count on the positive or negative input edge and select any of the SOURCE inputs, GATE inputs or Crystal Scaler outputs (F1-F5). Note that this lets you connect several counters to the same source or a standard frequency input just through software! For cascading counters, you can connect to the terminal count output of the next lower counter; for example, for 32 or 48 bit counters, etc.

CM3-7 control how the counter will operate. Essentially each bit performs a specific function.

CM0-2 control the terminal count output characteristics. It may be permanently low, high impedance, active low pulse, active high pulse, or toggled on terminal count.

3.5 DIGITAL I/O

Totally separate from the AMD-9513 counter, are two 8-bit digital I/O ports with latches. These I/O ports can be used for any purpose independent of the counter.]

The port at Base address +2 provides eight bits of TTL and TTL/LS compatible digital input. This input port uses a transparent 8-bit latch (74LS373), while the STROBE line is high, data passes through the latch. Data present when the STROBE line is taken low will be latched and held as long as the STROBE line remains low.

The port at Base address +3 is an 8-bit TTL digital output port. Each output can sink up to 8mA and can drive five standard TTL loads or 20 low-power schottky TTL loads.

3.6 INTERRUPT INPUT

A rising edge-triggered flip-flop (buffered by Schmitt trigger inverters) indicates the occurrence of an interrupt on the INTERRUPT INPUT signal at Pin 1. The interrupt can be read at Base +5 and is cleared by a write to Base +4. A pending interrupt will cause an interrupt to the computer only if an Interrupt Level was selected during Programmable Option Select (Interrupts 3, 5, 7, 9, 10, 11, 12, 15, or disabled are available choices) and if the INTERRUPT ENABLE (Pin 2) is logic low. Both INTERRUPT INPUT and INTERRUPT ENABLE are pulled to logic high with a 10 Kohm resistor. The UCCTM-05 Micro Channel interrupt circuitry fully conforms to the IBM requirements for level sensitive interrupt sharing (refer to the PS/2 Technical Reference Manual for more details).

Typically the counter outputs can be jumpered into the INT.IN, and the INT.ENABLE might be controlled by one of the digital port outputs OP0-7. This would allow periodic interrupts. Alternatively the input can be used for other purposes such as transferring data into and out of the computer on external events.

■ ■ ■

Chapter 4

CALIBRATION

The CTM-05 Board does not require calibration. The Board's timebase is crystal-controlled and has no frequency adjustment. The following specifications determine the accuracy of the timebase.

FREQUENCY: 1.0000Mhz

FREQUENCY STABILITY: ±0.01% (0 to 70 °C; +5V ±0.5V)

The crystal accuracy can be monitored with a frequency counter at the FOUT signal (Pin 30) and digital common (Pin 11) by setting FOUT to 1 MHz with the following BASIC commands.

```
OUT BASE+1,23

OUT BASE,0

OUT BASE,1
```

■ ■ ■

❑

❑

❑

Chapter 5

PROGRAMMING

5.1 GENERAL

The CTM-05 is programmable at the lowest level using input and output instructions. In BASIC these are the INP(X) and OUT X,Y functions. Assembly language and most other high level languages have equivalent instructions.

To simplify program generation, the distribution software contains the I/O driver routine *CTM5.BIN*. This routine is accessible from BASIC using a single-line CALL statement, as follows:

```
100  CALL CTM5 (MD%, DIO%(0), FLAG%)
```

The 11 operating modes (MD%) of the CAL routine select most of the uCCTM-05 functions, transfer counter data to and from BASIC variables (array DIO%(0)), check for errors (FLAG%), and perform complex operations such as measuring frequency or transferring the counter contents to memory on periodic interrupts. Note, however, that BASIC has no interrupt or DMA processing functions, and so-called *background* data collection using these methods is available only by using the CALL routines.

Because of the large number of 9513 operating modes, the CTM5.BIN driver is a compromise between simplicity and flexibility. The driver can perform many of the more common operations that CTM-05 is likely to be used for, but it will require programming with INP and OUT statements for some of the less common operations.

5.2 LOADING THE CTM5.BIN DRIVER ROUTINE (BASIC)

To use CALL routine *CTM5.BIN*, you must first load it into memory. Avoid loading it over any part of memory that is in use by another program (BASIC, print spoolers, or Disk-RAM). If you interfere with another program's use of memory, CALL routine will not work and your PC will probably hang up (Turn off power and wait a few seconds before turning on again). Note that the information given in this section is general; it applies to loading any CALL or USR routine and supplements the limited information in Appendix C of the "IBM BASIC MANUAL". For further enlightenment on this subject see "The 8088 Connection" by Dan Rollins, page 398 of "Byte" magazine, July 1983.

You have two options depending on the size of available memory: (1) loading outside BASIC's normal workspace or (2) contracting BASIC's workspace and loading at the end of the reduced workspace. The second method is somewhat more complicated but required if you have limited memory and if BASIC is initially unable to find 64K of workspace (the maximum it can use). If BASIC is using its maximum 64K, you get the following message on power-up or from DOS by entering **BASIC(A)** .

```
BASICA      The IBM Personal Computer Basic
(DOS 3.0)   Version A3.00 Copyright IBM Corp.  1981, 1982, 1983, 1984
             60451 Bytes free
             Ok
```

When the number of free memory bytes is less than that shown above for the version of BASIC in use, your PC's memory is already fully used, and BASIC adjusts to this condition by using less than its possible 64K maximum. If this is the case, you must load the CALL routine by further forced contraction of the BASIC workspace and loading the routine at the end of the newly defined workspace. CTM-5.BIN occupies about 1.8K bytes, but for simplification let's clear a 2K space for it.

Step 1 is to work out how much memory BASIC is actually using. Let's assume you see "XXXXX Bytes free" after loading BASIC as above. Now subtract 4096 (4K) from the above number. The result is the maximum amount of working space that can be allocated to BASIC with the CALL routine loaded. (You always have the option to allocate less working space if you wish.) Let's call the size of the workspace WS. This space can be allocated either when loading BASIC from DOS, as follows:

```
BASIC(A) /M:WS
```

or usually more conveniently by using CLEAR at the beginning of a program, as follows:

```
xxx10 CLEAR, WS
```

Next, we need to know what segment BASIC is occupying in memory. This can be found from the contents of memory locations &H511 and &H510 which hold the current BASIC segment which we can call SG. SG can be determined as follows:

```
xxx20 DEF SEG = 0      'define code segment = 0000 before
                                'reading absolute addresses
                                '0000:0510 & 0000:0511

xxx30 SG = 256*PEEK(&H511) + PEEK(&H510)
```

The segment address at which we can now load the CALL routine will be at the end of the working space, that is

```
xxx40 SG = WS/16 + SG        'remember segment addresses are on
                                    '16-bit boundaries
```

The routine can now be loaded as follows:

```
xxx50 DEF SEG = SG
xxx60 BLOAD "CTM5.BIN",0     'loads routine at SG:0000
```

A BLOAD must be used as we are loading a binary (machine language) program. Once loaded, the CALL can be entered as many times as needed in the program after initializing the call parameters MD%, DIO%, FLAG% prior to the CALL sequence as follows:

```
xxx70 DEF SEG = SG
xxx80 CTM5 = 0
xxx90 DIM DIO%(9)
xx100 CALL CTM5 (MD%, DIO%(0), FLAG%)
```

Note that CTM5 is a variable that specifies the memory offset of the starting address of the CALL routine from the current BASIC segment. We have chosen CTM5 as a name as it makes CALL CTM5 easy to remember and would distinguish it from any other CALL to some other routine that might be in the same program. This is purely a matter of choice and programming style, just easier to remember than writing CALL X (.....) or CALL AB(3) (.....) etc.. The variable CTM5 is the offset (actually zero) from the current segment as defined by the last DEF SEG statement that tells your BASIC interpreter where the CALL routine is located. Be careful that you do not inadvertently redefine the current segment somewhere in a program before entering the CALL. It is good practice to immediately precede the CALL statement by the appropriate DEF SEG statement (the same one you preceded your BLOAD with) even at the cost of duplication. This precaution can save a lot of wasted time and frustration from crashing your computer!

Another important detail is that CLEAR sets working space from the bottom of the BASIC working area up whereas we must set aside space for our subroutine from the top of available memory down. If we attempt to CLEAR more space than is actually available, we will end up loading our routine over the end of the BASIC program, data space and stack and will hang up the computer. Be careful this does not happen inadvertently if you are memory limited and later load BASIC with DEBUG or some other co resident program without making a compensating reduction in the workspace (WS) declaration in the CLEAR statement. If possible, setting up a workspace that is a considerable amount less than the maximum available is a simple precaution.

The second option is somewhat simpler to follow and applies when you have plenty of memory and are able to load the CALL routine outside the BASIC workspace. In this case, choose a segment that has 2K bytes clear at its beginning. For example we might choose &H2000 which is at 128K on a machine with a minimum of 192K memory. Then proceed as follows:

```
xxx10 DEF SEG = &H2000        'Sets up load segment
xxx20 BLOAD "CTM5.BIN",0      'Loads at 2000:0000
xxx30 CTM5 = 0
   ""
   ""
   ""
xxxyy CALL CTM5 (MD%, DIO%(0), FLAG%)
xxxzz etc.
```

An example of this approach is also contained in file LOADCALL.BAS. Before you try loading outside the workspace, be sure you really have an unused 2K of memory at 128K. You can change the DEF SEG statements in line xxx10 and experiment with loading the CALL routine at other locations. Usually any clash with another program's use of the same memory results in obliteration of some of the routine code and a failure to exit and return from the routine. The computer hangs up, and the only cure is to switch off, wait a few seconds and turn on the power again.

5.3 CALL STATEMENT FORMAT (BASIC)

Prior to entering the CALL, the DEF SEG=SG statement sets the segment address at which the CALL subroutine is located. The CALL statement for the CTM5.BIN driver must use the form:

```
xxxxx CALL CTM5(MD%, D%(0), FLAG%)
```

CTM5 is the address offset from the current segment of memory, as defined in the last DEF SEG statement. In all the examples, the current segment is defined to correspond with the starting address of the CALL routine. This offset is therefore zero and CTM5=0.

The three variables within brackets are known as the CALL parameters; their meaning depends on the Mode, as described in the following sections. On executing the CALL, the addresses of the variables (pointers) are passed in the sequence written to BASIC's stack. The CALL routine unloads these pointers from the stack and uses them to locate the variables in BASIC's data space so data can be exchanged. Four important format requirements must be met:

1. The CALL parameters are positional. The subroutine knows nothing of the names of the variables, just their locations from the order of their pointers on the stack. The parameters must always be written in the correct order:

   (mode, data, errors)

2. The CALL routine expects its parameters to be integer-type variables and will write and read to the variables on this basis.

3. You cannot perform any arithmetic functions within the parameter list brackets of the CALL statement. For example, the following is an **illegal** statement:

   CALL CTM5(MD%+2,D%(0) * 8,FLAG%)

4. You cannot use constants for any of the parameters in the CALL statement. For example, the following is an **illegal** statement:

   CALL CTM5(7,2,FLAG%)

   This must be programmed as

   XXX10 MD% = 7
   XXX20 DIO%(0) = 2
   XXX30 CALL CTM5 (MD%, DIO%(0), FLAG%)

Apart from these restrictions, you can name the integer variables what you want; the names in the examples are just convenient conventions. Strictly, you should declare the variables before executing the CALL.

5.4 USE OF THE CALL ROUTINE

The following subsections contain details and examples of using the CALL routine in all twelve CTM-05 Modes. The Modes are selected by the MD% parameter in the CALL as follows:

MODE (MD%)	FUNCTION
0	Initialize, set Master Mode Register and Base I/O address.
1	Set a Counter Mode Register.
2	Multiple counter control commands, arm, load, latch, etc.
3	Load a selected Counter Load Register data.

4	Read a selected Counter Hold Register.
5	Read Digital Input Port IP0-7.
6	Write Digital Output Port OP0-7.
7	Latch counter(s) and store data on interrupt.
8	Return status of interrupts.
9	Unload Interrupt data from memory and transfer to BASIC array variable.
10	Measure frequency of up to nine inputs.
11	Latch Counter(s) and store Segment and Offset on Interrupt.

5.5 MODE CALL DESCRIPTIONS

MODE 0 - Initialize

Mode 0 checks that the base I/O address is in the legal range of 256 - 1020 (Hex 100 - 3FC) for the IBM P.C.. If not, an error exit occurs. If OK, the Base I/O address is stored for use by other modes on re-entry to the CALL.

Mode 0 must be executed as an initializing step before any of the other modes are selected. Selecting any other mode without having entered mode 0 will give error code 1 as the driver will not be aware of the I/O location of the CTM5.

After storing the base I/O address, the 9513 Master Mode Register is loaded according to the contents of DIO%(1)-DIO%(5). A few default conditions are assumed:

1. MM15=1 scaler set to BCD. Since a 1MHz crystal is used as standard, BCD scaling gives round number sub-multiples:

   F1 = 1MHz
   F2 = 100KHz
   F3 = 10KHz
   F4 = 1KHz
   F5 = 100Hz

2. MM14=1 data pointer automatic increment disabled. Automatic increment is not used by the driver.
3. MM13=0 8 bit data bus is required by the hardware.
4. MM12=0 Fout is permanently on.

The remaining master mode register bits are controlled by the input variables.

Entrance data is as follows:

DIO%(0) = Base I/O address (100H - 3FCH)

DIO%(1) = Fout divider ratio (0 - 15) 0 = /16 otherwise N = /N]

DIO%(2) = Fout source (0 - 15)

0 = F1

1 = SOURCE 1

2 = SOURCE 2

3 = SOURCE 3

4 = SOURCE 4

5 = SOURCE 5

6 = GATE 1

7 = GATE 2

8 = GATE 3

9 = GATE 4

10 = GATE 5

11 = F1

12 = F2

13 = F3

14 = F4

15 = F5

DIO%(3) = Compare 2 disable/enable (0/1)

DIO%(4) = Compare 1 disable/enable (0/1)

DIO%(5) = Time of day mode control (0 - 3)

Exit data is as follows:

DIO%(0-9) - Unchanged

The following error codes apply to MODE 0:

FLAG% = 0 (no error, OK)
= 2 (mode number out of range, <0 or >11)
= 3 (base address out of range <256 or >1020)
= 11 thru 19 (DIO%(1) thru (DIO%(9) out of range): DIO%(2) wrong gives error # 12

Note that error 3 will occur if you have specified an I/O address that is less than 256 (Hex 100) or greater than 1020 (Hex 3FC). I/O addresses below Hex 100 are all used internally by devices on the IBM P.C. system board and would always cause an address conflict with CTM-5 and addresses above Hex 3FF are not exclusively decoded by other peripherals on the IBM PC.

MODE 1 - Set A Counter Mode Register

Mode 1 is used to configure each individual counter by setting the associated mode register. After mode 0, mode 1 usually is a necessary second level of initialization.

Entrance data:

DIO%(0) =Counter number (1 - 5)

DIO%(1) = Gating control (0 - 7)

0 - No gating
1 - Active high level TCN-1
2 - Active high level GATE N+1
3 - Active high level GATE N-1
4 - Active high level GATE N
5 - Active low level GATE N
6 - Active high edge GATE N
7 - Active low edge GATE N]

DIO%(2) = Count edge positive/negative (0/1)

DIO%(3) = Count source selection (0 - 15)

0 - TCN-1
1 - SOURCE 1
2 - SOURCE 2
3 - SOURCE 3
4 - SOURCE 4
5 - SOURCE 5
6 - GATE 1
7 - GATE 2
8 - GATE 3
9 - GATE 4
10 - GATE 5
11 - F1
12 - F2
13 - F3
14 - F4
15 - F5

DIO%(4) = Disable/enable special gate (0/1)

DIO%(5) = Reload from load/Reload from load or hold (0/1)

DIO%(6) = Count once/count repetitively (0/1)

DIO%(7) = Binary count/B.C.D. count (0/1)

DIO%(8) = Count down/count up (0/1)

DIO%(9) = Output control (0 - 5, except 3)

0 - Inactive, output low
1 - Active high terminal count pulse
2 - Terminal count toggled
3 - Illegal
4 - Inactive, output high impedance
5 - Active low terminal count pulse

Exit data:

DIO%(0-9) - Unchanged

The following error codes apply to MODE 1:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)
= 10 thru 19 (DIO%(0) thru (DIO%(9) out of range) e.g. DIO%(2) wrong gives Error 12

MODE 2 - Multiple Counter Control Commands

MODE 2 allows you to perform operations such as loading, latching and saving, enabling, and disabling on individual or multiple counters simultaneously. A linear select using DIO%(1) thru DIO%(5) for Counters 1 - 5 performs the counter addressing. This is a powerful feature of the 9513. If you wish to operate on a counter, the corresponding DIO%() variable should be set to 1; otherwise it should be set to 0. The chosen command (1-6) is set by the value of DIO%(0). Note the following:

1. The terms ARM and DISARM are synonymous with ENABLE and DISABLE. A disarmed counter will not count or respond to its clock and gate inputs.
2. Each counter has an associated Load and a Hold Register. To load a counter, use MODE 3 to load its Load Register. The Load Register(s) can then be transferred into the counter(s) using Commands 2 or 3 of this MODE. Similarly, counter contents can be transferred into the Hold Register(s) using Commands 4 or 5 of this MODE. Note that this is a simultaneous transfer for all selected counters, and in the case of Command 5, the counting process is not disturbed, allowing you to read any of the counters simultaneously "on the fly." Finally, the contents of the Hold Registers can be read at leisure using MODE 4.
3. The operation of unselected counters is not disturbed in any way by operations on selected counters.

Entrance data:

DIO%(0) = Command (1 - 6) as follows:
1 - Arm selected counter
2 - Load source to counter
3 - Load and arm counter
4 - Disarm and save counter
5 - Latch counter to Hold Register
6 - Disarm counter

DIO%(1) = Select Counter 1 (0/1)

DIO%(2) = Select Counter 2 (0/1)

DIO%(3) = Select Counter 3 (0/1)

DIO%(4) = Select Counter 4 (0/1)

DIO%(5) = Select Counter 5 (0/1)

DIO%(6-9) - Not used, don't care

Exit data:

DIO%(0-9) - Unchanged

The following error codes apply to MODE 2:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)
= 10 (Command Number out of range, <1 or >6)
= 11 (Counter 1 select not 0 or 1)
= 12 (Counter 2 select not 0 or 1)
= 13 (Counter 3 select not 0 or 1)
= 14 (Counter 4 select not 0 or 1)
= 15 (Counter 5 select not 0 or 1)]

MODE 3 - Load Counter Load Register

MODE 3 is used to place data in any selected counter's Load register. Note that this mode does not physically load the counter until MODE 2 performs a Load and Arm (Enable) or Load command, which transfers data from the Load Register into the counter. This method allows counters to be simultaneously loaded and started even though data enters sequentially into each Load Register.

Entrance data:

DIO%(0) = Counter number (1 - 5)

DIO%(1) = Load data (-32768 to +32767)

DIO%(2 - 9) - Not used, don't care

Exit data:

DIO%(0-9) - Unchanged

The following error codes apply to MODE 3:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)
= 10 (counter number out of range, <1 or >5)

MODE 4 - Read Selected Counter Hold Register

MODE 4 allows the reading of a selected counter's Hold Register. Note that this MODE does **not** read the contents of the counter directly. A counter's contents must be transferred to its Hold Register using MODE 2 before entering MODE 4 to read the counter contents indirectly. There is no direct method of reading a counter.

Entrance data:

DIO%(0) = Counter number (1 - 5)

DIO%(1) = Data Read variable, value does not matter

DIO%(2 - 9) - Not used, don't care

Exit data:

DIO%(0) = Counter number (1 - 5)

DIO%(1) = Counter data (-32768 to +32767)

DIO%(2 - 9) =Unchanged

The following error codes apply to MODE 4:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)
= 10 (counter number out of range, <1 or >5)

MODE 5 - Read The Digital Input Port

MODE 5 allows you to read the state of the Digital Input Port IP0-IP7. This port is consists of a 74LS373N transparent latch and has a hardware strobe input on Pin 21 of the rear connector. When the STROBE INPUT is high, data at the inputs may be read directly; when the STROBE INPUT is taken low, data at IP1-IP7 is latched and reading the port yields the latched data regardless of the state of the inputs. 8-bit data (range 0-255) is returned in DIO%(0).

Note that the Digital Input Port is entirely independent of the 9513 counter. MODE 5 is equivalent to a Basic INP (Base +2) instruction.

Entrance data:

DIO%(0) = Data Read variable, value does not matter

DIO%(1 - 9) - Not used, don't care

Exit data:

DIO%(0) = Input port data (0 - 255)

DIO%(1-9) - Unchanged

The following error codes apply to MODE 5:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)

MODE 6 - Write To Digital Output Port

MODE 6 allows you to write data to the 8-bit Digital Output Port OP0-OP7. Data should be in the range 0 to 255 decimal, corresponding to eight binary bits. MODE 6 performs an equivalent function to Basic's OUT BASE + 3, DIO%(0). The output port is entirely independent of the 9513 counter.

Entrance data:
DIO%(0) = Output data, range 0 - 255 (8 bit)

DIO%(1 - 9) =Not used, don't care

Exit data:

DIO%(0-9) =Unchanged

The following error codes apply to MODE 6:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)
= 10 (output data out of range, <0 or >255)

MODE 7 - Latch Counters & Save On Interrupt

MODE 7 is complex in that it transfers selected counter contents "on the fly" to buffer memory each time an interrupt occurs. A typical application might drive the interrupt input from the FOUT pin or a counter in Divide-By-N MODE, so that interrupts are generated at a constant rate. Any combination of the remaining counters can be set up to transfer their contents on each interrupt to individual buffer areas outside BASIC's workspace. This is especially useful when you need to measure the change in frequency versus time and also accumulate the total count. On each interrupt, the counters are simultaneously latched and data transferred to a selected segment (see MODE 11 to select offset). The next interrupt transfers data to (memory address + 2) so that a series of words builds up in memory to make a "snapshot" of the counter contents versus time. Each counter is allocated an individual segment of memory that may be up to 64K bytes in length, sufficient for 32,767 interrupts (a 16-bit counter uses 2 bytes or a word to store its data). The DIO%(1) - DIO%(5) variables control allocation of the buffer segment for each counter. Segments should be chosen outside BASIC's workspace to avoid writing over program/stack area and causing a crash.

If you enter a value of -1 in any of the Counter parameters (DIO%(1) - DIO%(5)), you must call MODE 11 with each counter having a memory segment and offset to dump data.

Before selecting MODE 7, decide on the Interrupt Level you wish to assign the CTM-05 and place the Interrupt Jumper accordingly (on Jumper Block J2).

MODE 7 initiates the following sequence of actions:

1. Loads interrupt vectors into memory for the selected level and stores any old vectors for subsequent automatic restitution at the end of interrupts.
2. Enables interrupt handler routine.
3. Initializes 8259 interrupt controller and enables 8259 interrupt mask register for the selected level. Interrupts are generated by a low to high transition on the INTERRUPT INPUT (Pin 1). This input is enabled when the INTERRUPT ENABLE (Pin 2) is held low.
4. Selects which counters are to be dumped on interrupt according to DIO%(1) thru DIO%(5), which control the dump segments in memory for each counter. If any of these variables is zero, dumping for that counter is be disabled.
5. Performs the number of interrupts (up to 32,767) set by DIO%(0) and then disables further interrupts and restores old vectors.

Notes on the hardware operation of MODE 7:

A. After setting interrupts running, exit takes place from MODE 7 to the following program. The data continues to be collected as a "background" operation. The progress of this operation can be monitored using MODE 8. The foreground program can be analyzing/manipulating data as it is collected.]

B. Because of the finite time that the interrupt handler takes to execute, the interrupt rate should not exceed 4000 per second. As the rate increases, more and more of the processor time is taken servicing the interrupts and less is available for servicing foreground tasks until eventually no time at all is available for the foreground between operations and interrupts may be skipped.

C. Latency, or the uncertainty of the instant of time when the interrupt is serviced may cause small variations or jitter in the sampling intervals. A major contributor to this jitter are other interrupts in the PS/2, especially the timer interrupt occurring 18 times/s on Level 0. If necessary this interrupt can be suppressed by reading the 8259 interrupt mask register and disabling the timer interrupt. This can be performed by the following BASIC code before entering MODE 7:

```
xxx00 IMR% = INP(&H21)
xxx10 OUT &H21, (IMR% OR &H01)
```

As the disk drives use the timer for run up, it should be re-enabled after interrupts by:

```
xxx20 OUT &H21,IMR%
```

Other interrupts can be avoided by not using the keyboard or COM(n) ports during data collection. Suppressing interrupts will reduce latency to the variation of a few clock cycles or a few microseconds.

Entrance data:

DIO%(0) = Number of interrupts (1 - 32767)

DIO%(1) = Memory segment to dump data for Counter 1 (0 - 65535)

DIO%(2) = Memory segment to dump data for Counter 2 (0 - 65535)

DIO%(3) = Memory segment to dump data for Counter 3 (0 - 65535)

DIO%(4) = Memory segment to dump data for Counter 4 (0 - 65535)

DIO%(5) = Memory segment to dump data for Counter 5 (0 - 65535)

DIO%(6) = Start on IP0 disabled/enabled (0/1)

DIO%(7) = Interrupt level (2 - 7)

DIO%(8-9) =Not used, value does not matter

NOTE: If any of the dump segments DIO%(1-5) is set to zero, then dumping of that counter's data is disabled. This provides a means of selecting which counter's data will be stored on interrupt.

Exit data:

DIO%(0-9) - Unchanged

NOTE: Hardware gating of the interrupt operation may be performed with the INTERRUPT ENABLE input, Pin 2.

The following error codes apply to MODE 7:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)
= 10 (interrupt count out of range, <=0)]

MODE 8 - Return Status Of Interrupts

MODE 8 provides a means of determining the progress of an interrupt operation initiated by MODE 7. DIO%(0) returns the status in terms of whether the interrupts are still active or finished, and DIO%(1) returns the current word count (number of interrupts).

Entrance data:

DIO%(0-9) = Value irrelevant

Exit data:

DIO%(1) = Interrupt active/finished (1/0)

DIO%(2) = Current word count

DIO%(2-9) - Unchanged

The following error codes apply to MODE 8:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)

MODE 9 - Transfer Data During/After Interrupt

MODE 9 is a general-purpose block-transfer routine that moves any number of words from any position in any memory segment to a suitably dimensioned integer array in BASIC. Data may be transferred in small blocks, piece-by-piece, and when it is not possible to dimension a large integer array (to hold 32,767 words because of limitations in BASIC's workspace). MODE 9 is faster than using BASIC PEEK's, which perform the same function. MODE 9 is usually used to retrieve data after Interrupt MODE 7 has placed data in memory.

Entrance data:
DIO%(0) - Number of words to transfer (1 - 32767)

jDIO%(1) - Starting word number (0 - DIO%(0))

DIO%(2) - Memory segment to transfer from

DIO%(3) - Starting integer array element address (offset) to transfer data to.

DIO%(4-9) - Not used, value does not matter.

NOTE: The pointer to the starting element in each array is provided by BASIC's VARPTR function, as follows:

```
DIO%(3) = VARPTR(ARRAY%(N))
```

It is the programmer's responsibility to insure that integer data arrays are adequately dimensioned to receive the data. Overrunning the array may have strange effects and cause program crashes.

Exit:

DIO%(0-9) - Unchanged. Data transferred to selected array

The following error codes apply to MODE 9:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)
= 10 (number of words <=0)
= 11 (starting word number <0)]

MODE 10 - Measure Frequency

MODE 10 uses several of the unique features of the AMD9513 to measure up to nine external frequency inputs. The TTL-compatible input signals are applied to any or all of SOURCES 1 - 5 or GATES 1 - 4.

The measurement technique makes use of a pair of counters. Counter 4 operates as a timebase in the toggled MODE. Its input is internally gated to the F4 crystal divider source. On the CTM-05 in BCD (default) this is a 1KHz signal. This timebase counter is loaded in Countdown MODE with the gating interval, so that the toggled output is alternately high for the gate interval and then low for the gate interval. Counter 5 is used to accumulate pulses during the gate interval. Its gate (GATE 5 - Pin 12) must be connected externally to the output of Counter 4 (COUNTER 4 OUTPUT - Pin 32) and its clock input can select from any of the nine SOURCE or GATE inputs. While its gate is high, Counter 5 accumulates input pulses. The state of Counter 4's output and hence Counter 5's gate is sensed in the routine by reading the 9513 status register through the control port.

A measurement is performed as follows:

1. Set the MODE registers of Counters 4 and 5.
2. Load and continuously run the Timebase Counter 4.
3. Wait for the Timebase Counter output to go from high to low by reading the Status Register.
4. Load counter 5 with zero. Counter 5 is in the count up MODE and is now disabled as its gate is low.
5. Wait for Counter 5's gate to go high and then low again by reading the Status Register for Counter 4's output. Counter 5 now contains the accumulated count in the gating interval.
6. Read Counter 5 and transfer data to DIO%(2).
7. Return to BASIC.

Note the following features and limitations:

A. The gate interval can be from 1ms to 32.767s as set by DIO%(0).

B. The accumulating counter has a 16-bit resolution (1 part in 65,535). For a given frequency range of input, the gate interval should be chosen to use this resolution effectively (that is, don't use a 1mS gate with a 10KHz signal (10 counts) or a 1s gate with a 5MHz signal (5,000,000 counts)). In the event of a counter overflow, the counter will continue counting but there is no way of ascertaining the number of times it overflows.

C. Due to the measurement algorithm used, it can take up to three gate intervals to return a result. With longer gate intervals (10 seconds), your computer may appear to have hung up when it is simply waiting for the result.

D. Nine separate frequency sources can be connected to the GATE and SOURCE inputs of the CTM5 at the same time. Measurements are, however, performed on one source at a time. In effect, the 9513 is used to multiplex the input signals as well as count them. The selected source is controlled by DIO%(1). Counters 1 thru 3 are unaffected by MODE 10 and can be used for other purposes.

E. COUNTER 5 GATE must be externally jumpered to COUNTER 4 OUTPUT. This requires an external jumper on the CTM-05 connector from Pin 12 to Pin 32.

F. The crystal oscillator precision is better than 0.01% after adjustment of the trimmer capacitor; otherwise it will be within 0.1% (see Calibration in the manual). The maximum frequency input of the 9513 with optimum 50% duty cycle is 7MHz.

G. The manufacturer reminds you that this is a method (not necessarily the best method) of using the 9513 to measure frequency. You can modify this MODE to cascade accumulator counters to 32 bits for more precision, etc. There are many possibilities.

Entrance data:

DIO%(0) = Gate interval in ms (1 - 32767)

DIO%(1) = Selects input signal source (1 - 9)

1 = SOURCE 1
2 = SOURCE 2
3 = SOURCE 3
4 = SOURCE 4
5 = SOURCE 5
6 = GATE 1
7 = GATE 2
8 = GATE 3
9 = GATE 4

DIO%(2-9) - Not used, value does not matter]

Exit data:

DIO%(0-1) - Unchanged

DIO%(2) - Counts accumulated in gating interval

DIO%(3-9) - Unchanged

The following error codes apply to MODE 10:

FLAG% = 0 (no error, OK)
= 1 (Base Address unknown)
= 2 (MODE number out of range, <0 or >11)
= 10 (gate interval out of range <1 or >32,767)
= 11 (source input out of range <1 or >9)

MODE 11 - Latch Counters & Save On Interrupt; Dump Data To Selected Offset & Segment

MODE 11 is the same as MODE 7 except that the caller can pass both segment and offset for data that is to be dumped (MODE 7 can pass only a segment). MODE 11 may be used only after MODE 7 has been called.

MODE 11 uses the following MODE 7 parameters:

DIO%(0) =Number of interrupts (1-32767).

DIO%(6) =Start on IP0 Disabled/Enabled (0/1).

DIO%(7) =Interrupt Level (2-7).

In addition, the caller must pass a value of -1 in any one parameter (DIO%(1) - DIO%(5)) in MODE 7 to indicate that an offset and segment will be passed in MODE 11.

Read Section 5.5 (MODE 7) for a general description of Mode 11.

Entrance data:

DIO%(0) =Memory offset for a Counter 1 data dump.

DIO%(1) =Memory segment for a Counter 1 data dump.

DIO%(2) =Memory offset for a Counter 2 data dump.

DIO%(3) =Memory segment for a Counter 2 data dump.

DIO%(4) =Memory offset for a Counter 3 data dump.

DIO%(5) =Memory segment for a Counter 3 data dump.

DIO%(6) =Memory offset for a Counter 4 data dump.

DIO%(7) =Memory segment for a Counter 4 data dump.

DIO%(8) =Memory offset for a Counter 5 data dump.

DIO%(9) =Memory segment for a Counter 5 data dump.

NOTE: Setting any of the dump segments (DIO%(1), DIO%(3), DIO%(5), DIO%(7), or DIO%(9)) to zero disables that counter's data dump. This features thus provides a means for selecting the counter data to be stored on interrupt.

Exit data:

DIO%(0-9) =Unchanged.

NOTE: Hardware gating of the interrupt operation may be performed with the Interrupt Enable Input (Pin 2).

Error codes for MODE 11 are as follows:

Flag% = 0 (no error, OK).
= 1 (Base Address unknown).
= 2 (MODE number out of range, < 0 or > 11).
= 10 (Interrupt count out of range).
= 17 (Interrupt Level not between 2 and 7).
= 21 (Counter 1 segment wraparound).
= 22 (Counter 2 segment wraparound).
= 23 (Counter 3 segment wraparound).
= 24 (Counter 4 segment wraparound).
= 25 (Counter 5 segment wraparound).

5.6 SUMMARY OF ERROR CODES

If for any reason the FLAG% variable is returned non-zero, then an error has occurred in the input of data to the CALL routine. Checking for valid data occurs first in the routine and no action occurs if an error condition exists; an immediate return takes place with the error specified in the FLAG% variable.

A list of error codes follows:

ERROR	FAULT
1	Base Address unknown. Failure to initialize Base Address using MODE 0.
2	MODE number out of range. Specifying MD% less than 0 or greater than 10.
3	Base Address out of range. Invalid base I/O address. Valid addresses must be in range 256 thru 1020 (Hex 100 to 3FC).
10-19	These error codes apply to the data parameters being out of range for the particular MODE selected. If DIO%(4) is incorrect, Error Code 14 is generated; or, in general, Error Code 10 + N is generated when DIO%(N) is wrong. See each MODE for the specific error conditions.
21-25	These error codes apply only in MODE 11, when the offset added to the word count crosses the segment boundary.

Error detection after the CALL routine is easily implemented, as follows:

```
xxx10 CALL CTM5 (MD%, DIO%(0), FLAG%)
xxx20 IF FLAG% <> 0 THEN GOSUB yyyyy
.   .   .
.   .   .
yyyyy REM: Error handling subroutine
.   .   .
.   .   .
zzzzz RETURN
```

This program is useful while debugging a new program. With a suitable error handling subroutine, it can be left permanently in place in the program.

5.7 ASSEMBLY LANGUAGE PROGRAMS & CALLS IN OTHER LANGUAGES

For assembly language programmers, the fully commented source code for CTM5.BIN is on the CTM-05 software distribution disk, in a file named CTM5.ASM. This source listing an excellent starting point for adapting or customizing routines to special requirements. Any word processor can be used to modify CTM5.ASM to your specific requirements. It can then be assembled and turned into a loadable BASIC machine language subroutine. For detailed instructions on using the DOS UTILITIES and MACRO-ASSEMBLER for this, print out the file HOWTO.DOC on the distribution disk. Assembly language programming should be attempted only by experienced programmers who are fully conversant with the hardware and software features of the computer. The manufacturer cannot support modified drivers or software.

To facilitate use of the I/O driver CALL routines with compiled BASIC, the assembly object code file CTM5.OBJ is provided. This was assembled using the IBM Macro Assembler and may be linked to other object modules from compilers, etc.. When using the linker, the routine's public name is CTM5 (see next paragraph concerning use of BASIC COMPILER).

One quick fix to improve the speed of an interpreted BASIC program is to compile it using the BASIC COMPILER. When you compile a BASIC program the significance of the CTM5 in the CALL statement is no longer the same:

```
xxx10 CALL CTM5 (MD%, DIO%(0), FLAG%)
```

CTM5 is not interpreted by the compiler as a variable. It becomes the public name of the subroutine that you wish to call. Before compiling your program remove lines that BLOAD the CTM5.BIN routine and all DEF SEG statements that control the location of the routine. These are not required as the linker will locate the CTM5 routine in memory automatically. After compiling your program, run the linking session as follows:

```
LINK yourprog.obj + ctm5.obj
```

CTM5.OBJ is on your distribution disk for this purpose.

5.8 MULTIPLE CTM-05s IN ONE SYSTEM

What if you wish to operate more than one CTM-05 in a system? To avoid conflicts, each CTM-05 must have a different Base Address and a different interrupt level (if interrupts are used. If on a common level, each board's interrupt can be enabled only one at a time. Each board must also be assigned its own CALL routine. To do this, first load the CTM5.BIN routine at different locations in memory:

```
xxx10 DEF SEG = SG1
xxx20 BLOAD "CTM5.BIN",0
xxx30 SG2 = SG1 + 2048/16 'allow 2K for each routine
xxx40 DEF SEG = SG2
xxx50 BLOAD "CTM5.BIN",0
xxx60 SG3 = SG2 + 2048/16 'etc.  for other boards]
```

Now the CALL appropriate to each board can be entered as required. Note that each CALL is preceded by a DEF SEG appropriate to that board:

```
yyy10 DEF SEG = SG1
yyy20 CALL CTM51 (MD%, DIO%(0), FLAG%)
yyy30 DEF SEG = SG2
yyy40 CALL CTM52 (MD%, DIO%(0), FLAG%) 'etc.]
```

5.9 EXAMPLE PROGRAMS

The distribution disk contains several example programs:

COUNT1.BAS	A five channel up counter using BASIC commands only (Does not use CTM5.BIN).
LOADCALL.BAS	Examples of loading and initializing using the CTM5.BIN driver.

SETFREQ.BAS	Setting a counter to output a specific frequency.
FREQ.BAS	Measuring frequency of up to 9 sources using MODE 10 of the driver.
INT.BAS	Sampling frequencies using interrupts.
COUNT2.BAS	A program equivalent to COUNT1.BAS using the CTM5.BIN driver. This is an interesting comparison with COUNT1.BAS.
QBCNT2.BAS	A program equivalent to COUNT2.BAS using QuickBASIC.
MSCMCNT2.C	A program equivalent to COUNT2.BAS using Microsoft C.
TCMCNT2.C	A program equivalent to COUNT2.BAS using Turbo C.
MSFCNT2.FOR	A program equivalent to COUNT2.BAS using Microsoft FORTRAN.
MSPCNT2.PAS	A program equivalent to COUNT2.PAS using Microsoft PASCAL.
TP5CNT2.PAS	A program equivalent to COUNT2.BAS using Turbo PASCAL.
MSCL_INT.C	An example Interrupt program using Microsoft C (Large Model) and working through MODEs 7 and 11.

5.10 INTEGER VARIABLE STORAGE

BASIC stores data in integer variables of two bytes that hold signed data in the range -32,768 to +32,767. Frequently, you will need to load counters with unsigned (positive only), 16-bit (or 2 byte) data in the range 0 - 65,535. This section explains how to convert from the signed to the unsigned form and vice versa.]

The signed representation used in BASIC integer variables (% type) is 2's Complement. Each integer variable uses 16 bits or two bytes of memory, and 16 binary bits of data is equivalent to values from 0 to 65,535 decimal. The 2's Complement convention interprets the most significant bit as a sign bit so the range becomes -32,768 to +32,767 (a span of 65,535). Numbers are represented as follows:

	HIGH BYTE								LOW BYTE							
	D7	D6	D5	D4	D3	D2	D1	D0	D7	D6	D5	D4	D3	D2	D1	D0
+32,767	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
.			.		.						.		.			
+10,000	0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0
.			.		.						.		.			
+1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
.			.		.						.		.			
-10,000	1	1	0	1	1	0	0	0	1	1	1	1	0	0	0	0
.			.		.						.		.			
-32,768	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	^															

Sign Bit
1 if negative, 0 if positive

Integer variables are the most compact form of variable storage and so to conserve memory and disk space and optimize execution speed, all data exchange via the CALL is through integer type variables. This poses a programming problem when handling unsigned numbers in the range 32,768 to 65,535.

If you wish to input or output an unsigned integer greater than 32,767 then it is necessary to work out what its 2's Compliment signed equivalent is. As an example, assume we want to load a 16 bit counter with 50,000 decimal. An easy way of turning this to binary is to enter BASIC and execute PRINT HEX$(50000). This returns C350 or binary:

50,000 (Hex C350) Binary 1100 0011 0101 0000

Since the most significant bit is 1, this is stored as a negative integer and in fact the correct integer variable value is 50,000 - 65,536 = -15,536. The programming steps for switching between integer and real variables for representation of unsigned numbers between 0 and 65,535 is therefore:

From real variable N (0 <= N <= 65,535) to integer variable N%

```
xxx10 If N<+32767 then N% = N ELSE N% = N - 65536]
```

From Integer variable N% to real variable N:

```
xxx10 if N% >= 0 then N=N% else N = N% + 65536]
```

■ ■ ■

Appendix A

Instructions For PCF-CTM05 Callable Driver

Contents

■ ■ ■

PART 1: INTRODUCTION

1.1 OVERVIEW

The PCF-CTM05 software is for Pascal, C, and Fortran programmers writing data acquisition and control routines using CTM-05 boards. PCF-CTM05 supports all memory models for the following languages;

- Microsoft C (V4.0 - 6.0)
- Microsoft QuickC (V1.0 - 2.0)
- Borland Turbo C (V1.0 - 2.0)
- Microsoft PASCAL (V3.0 - 4.0)
- Borland Turbo PASCAL (V3.0 - 5.0)
- Microsoft FORTRAN (V4.0 - 4.1)
- QuickBASIC (V4.0 & higher)
- GW, COMPAQ, and IBM BASIC (V2.0 & higher)

The PCF-CTM05 consists of several assembly language drivers for the various supported languages along with example programs for each language. This manual is structured to illustrate memory model usage for each of the above languages and to include a brief example program at the end of each language section. Full source listings are included on the supplied disk.

This manual is not an introduction or operating guide to the supported CTM-05 board. You should be familiar with the boards' various operating MODES, PARAMETERS, and ERROR codes before attempting PCF-CTM05 implementation. Refer to the main chapters of this manual.

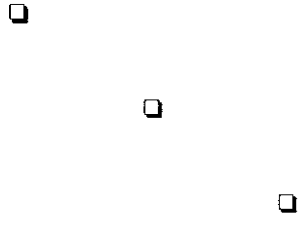
1.2 IMPLEMENTATION

Software drivers in the PCF-CTM05 packages support the CTM-05. As such, we urge you to become familiar with the board you are using before working with this interface package. Example programs herein do not assume any knowledge of these boards since the programs are general in nature and do not actually implement features of any specific board. They are limited to the actual language interface for the various languages supported.

In the following chapter, each interface driver (implemented via a CALL statement) consists of three position-dependent parameters. These are MODE, ARGUMENT (or PARAM), and FLAG, as follows:

MODE	Type of function to be executed by the CTM-05
PARAM	Function dependent arguments required for execution
FLAG	Error number, if any, corresponding to selected MODE

■ ■ ■



2.1 MICROSOFT C (V4.0 - 6.0) & QUICKC (V1.0 - 2.0)

Small Model

Model:	Small ("/AS") switch on command line
Passes:	word size pointers (offset, no DS register)
Sequence:	Arguments Passed Right to Left
Default Call Convention:	Arguments Passed by Value (Passing pointers to a subroutine is considered pass-by-value convention)

Example

'C' Call: **`mscs_CTM5 (&Mode, Params, &Flag);`**

'C' Declaration: **`extern void mscs_CTM5(int*,int*);`**

.ASM Subroutine:
The following assembly code shows how the driver handles user arguments:

```
_mscs_CTM5 proc near
        push bp                 ; save base pointer
        mov bp,sp               ; save stack pointer
        . ; [bp+4] holds offset of Mode
        . ; [bp+6] holds offset of Params
        . ; [bp+8] holds offset of Flag
        . ; Program execution here
        . ;
        . ;
        pop bp                  ;restore bp & sp prior to exit
        ret   ;return
_mscs_CTM5 endp
```

Other:

This information is provided for those wishing to create their own drivers:

- _mscs_CTM5 is declared "PUBLIC" in the .ASM file
- mscs_CTM5 is declared "extern" in the "C" file
- The .ASM file contains the ".model small" directive (MASM & TASM only)
- Add leading underscore "_" to all mscs_CTM5 occurrences in .ASM file
- mscs_CTM5 is a near call
- mscs_CTM5 must be in a segment fname_TEXT (where fname is the name of the file where mscs_CTM5 resides) if .ASM file contains mixed model procedures.

Medium Model

Model:	Medium ("/AM") switch on command line
Passes:	Word-size pointers (offset, no DS register)
Sequence:	Arguments Passed Right to Left
Default Call Convention:	Arguments Passed by Value

Example

'C' Call: `mscm_CTM5 (&Mode, Params, &Flag);`

'C' Declaration: `extern void mscm_CTM5(int*,int*,int*);`

.ASM Subroutine:
The following assembly code shows how the driver handles user arguments:

```
_mscm_CTM5 proc far          ; far CALL (dword return address)
        push bp              ; save base pointer
        mov bp,sp            ; save stack pointer
        . ; [bp+6] holds offset of Mode
        . ; [bp+8] holds offset of Params
        . ; [bp+10] holds offset of Flag
        . ; Program execution here
        . ;
        . ;
        pop bp               ;restore bp & sp prior to exit
        ret   ;return
_mscm_CTM5 endp
```

Other:

This information is provided for those wishing to create their own drivers:

- _mscm_CTM5 is declared "PUBLIC" in the .ASM file
- mscm_CTM5 is declared "extern" in the "C" file
- The .ASM file contains the ".model medium" directive (MASM & TASM only)
- Add leading underscore "_" to all mscm_CTM5 occurrences in .ASM file
- mscm_CTM5 is a far call
- mscm_CTM5 must be in a segment fname_TEXT (where fname is the name of the file where mscm_CTM5 resides), else Linker returns an error.

Large Model

Model:	Large ("/AL") switch on command line
Passes:	dword size pointers (offset and DS register)
Sequence:	Arguments Passed Right to Left
Default Call Convention:	Arguments Passed by Value

Example

'C' Call: `mscl_CTM5 (&Mode, Params, &Flag);`

'C' Declaration: `extern void mscl_CTM5(int*,int*,int*);`

.ASM Subroutine:
The following assembly code shows how the driver handles user arguments:

```
_mscl_CTM5 proc far          ; far CALL (dword return address)
        push bp              ; save base pointer
        mov bp,sp            ; save stack pointer
        . ; [bp+6] holds offset of Mode
        . ; [bp+10] holds offset of Params
        . ; [bp+14] holds offset of Flag
        . ; Program execution here
        . ;
        . ;
        pop bp               ;restore bp & sp prior to exit
        ret   ;return
_mscl_CTM5 endp
```

Other:

This information is provided for those wishing to create their own drivers:

- _mscl_CTM5 is declared "PUBLIC" in the .ASM file
- mscl_CTM5 is declared "extern" in the "C" file
- The .ASM file contains the ".model large" directive (MASM & TASM only)
- Add leading underscore "_" to all mscl_CTM5 occurrences in .ASM file
- Both code and data use dword (segment/offset) pointers
- mscl_CTM5 must be in a segment fname_TEXT (where fname is the name of the file where mscl_CTM5 resides), else Linker returns an error.

Microsoft 'C' Example

```
/*************************************************************************/
/* MSCEXAMPLE.C                                                          */
/* CTM-05 EXAMPLE OF MODE 0                                              */
/* USING MICROSOFT C MEDIUM MODEL                                        */
/*************************************************************************/

#include  "stdio.h"
#include  "conio.h"

extern mscm_CTM5(int*,int*,int*);   /* declare driver call */

main()
{
   int Mode, Flag, Params[15];

/* Initialize CTM-05 using Mode 0 */
```

```
   Mode = 0;
   Params[0]=768        /* Base Address of Board */
   Params[1]=10;        /* FOUT Divider Ratio of 10 */
   Params[2]=15;        /* FOUT Source */
   Params[3]=0;         /* Compare 2 Disabled */
   Params[4]=0;         /* Compare 1 Disabled */
   Params[5]=0;         /* Time Of Day Disabled */

   mscm_CTM5(&Mode, Params, &Flag);

   if(Flag !=0)
      {
        printf("\\n\\nMode 0 Error FLag = %d\\n",Flag);
      }

/* REMAINDER OF CODE */
.
.
.
}
```

2.2 BORLAND TURBO 'C' (V1.0 - 2.0)

Small Model

Model:	Small ("-ms") switch on command line
Passes:	word size pointers (offset, no DS register)
Sequence:	Arguments Passed Right to Left
Default Call Convention:	Arguments Passed by Value

Example

'C' Call: `tcs_CTM5(&Mode, Params, &Flag);`

'C' Declaration: `extern void tcs_CTM5(int*,int*,int*);`

.ASM Subroutine:
The following assembly code shows how the driver handles user arguments:

```
_tcs_CTM5 proc near
        push bp              ; save base pointer
        mov bp,sp            ; save stack pointer
        . ; [bp+4] holds offset of Mode
        . ; [bp+6] holds offset of Params
        . ; [bp+8] holds offset of Flag
        . ; Program execution here
        . ;
        . ;
        pop bp               ;restore bp & sp prior to exit
        ret   ;return
_tcs_CTM5 endp
```

Other:

This information is provided for those wishing to create their own drivers:

- _tcs_CTM5 is declared "PUBLIC" in the .ASM file
- tcs_CTM5 is declared "extern" in the "C" file
- The .ASM file contains the ".model small" directive (MASM & TASM only)
- Add leading underscore "_" to all tcs_CTM5 occurrences in .ASM file
- tcs_CTM5 is a near call
- tcs_CTM5 must be in a segment fname_TEXT (where fname is the name of the file where tcs_CTM5 resides), else Linker returns an error.

Medium Model

Model:	Medium ("-mm") switch on command line
Passes:	word size pointers (offset, no DS register)
Sequence:	Arguments Passed Right to Left
Default Call Convention:	Arguments Passed by Value

Example

'C' Call: `tcm_CTM5 (&Mode, Params, &Flag);`

'C' Declaration: `extern void tcm_CTM5(int*,int*,int*);`

.ASM Subroutine:
The following assembly code shows how the driver handles user arguments:

```
_tcm_CTM5 proc far          ; dword pointer return address
        push bp             ; save base pointer
        mov bp,sp           ; save stack pointer
        . ; [bp+6] holds offset of Mode
        . ; [bp+8] holds offset of Params
        . ; [bp+10] holds offset of Flag
        . ; Program execution here
        . ;
        . ;
        pop bp              ;restore bp & sp prior to exit
        ret   ;return
_tcm_CTM5 endp
```

Other:

This information is provided for those wishing to create their own drivers:

- _tcm_CTM5 is declared "PUBLIC" in the .ASM file
- tcm_CTM5 is declared "extern" in the "C" file
- The .ASM file contains the ".model medium" directive (MASM & TASM only)
- Add leading underscore "_" to all tcm_CTM5 occurrences in .ASM file

- tcm_CTM5 must be in a segment fname_TEXT (where fname is the name of the file where tcm_CTM5 resides), else Linker returns an error.

Large Model

Model:	Large ("-ml") switch on command line
Passes:	dword size pointers (offset and DS register)
Sequence:	Arguments Passed Right to Left
Default Call Convention:	Arguments Passed by Value

Example

'C' Call: `extern void tcm_CTM5(int*,int*,int*);`

'C' Declaration: `extern void tcl_CTM5(int*,int*,int*);`

.ASM Subroutine:
The following assembly code shows how the driver handles user arguments:

```
_tcl_CTM5 proc far          ; dword pointer return address
        push bp             ; save base pointer
        mov bp,sp           ; save stack pointer
        . ; [bp+6] holds offset of Mode
        . ; [bp+10] holds offset of Params
        . ; [bp+14] holds offset of Flag
        . ; Program execution here
        . ;
        . ;
        pop bp              ;restore bp & sp prior to exit
        ret   ;return
_tcl_CTM5 endp
```

Other:

This information is provided for those wishing to create their own drivers:

- _tcl_CTM5 is declared "PUBLIC" in the .ASM file
- tcl_CTM5 is declared "extern" in the "C" file
- The .ASM file contains the ".model large" directive (MASM & TASM only)
- Add leading underscore "_" to all tcl_CTM5 occurrences in .ASM file
- Both code & data use dword (segment/offset) pointers
- tcl_CTM5 must be in a segment fname_TEXT (where fname is the name of the file where tcl_CTM5 resides), else Linker returns an error.

Turbo 'C' Example

```
/*************************************************************************/
/* TCEXAMPLE.C                                                           */
/* CTM-05 EXAMPLE OF MODE 0                                              */
/* USING TURBO C MEDIUM MODEL                                            */
/*************************************************************************/

#include  "stdio.h"
#include  "conio.h"

extern tcm_CTM5(int*,int*,int*);     /* declare driver call

main()
{
   int Mode, Flag, Params[15];

/* initialize CTM-05 using Mode 0 */

   Mode = 0;
   Params[0]=768;       /* Base Address of Board */
   Params[1]=10;        /* FOUT Divider Ratio of 10.*/
   Params[2]=15;        /* FOUT Source */
   Params[3]=0;         /* Compare 2 Disabled */
   Params[4]=0;         /* Compare 1 Disabled */
   Params[5]0;          /* Time Of Day Disabled */

   tcm_CTM5(&Mode, Params, &Flag);
   if(Flag !=0)
      {
        printf("\\n\\nMode 0 Error FLag = %d\\n",Flag);
      }

/* REMAINDER OF CODE */

.
.
.
}
```

2.3 MICROSOFT PASCAL (V3.0 - 4.0)

Medium Model

Model:	Medium
Passes:	word size pointers (offset address only)
Sequence:	Arguments Passed Left to Right
Default Call Convention:	Arguments Passed by Value

Example

PASCAL Call: `Result:  =  msp_CTM5 (Var1, Var2, Var3);`

'C' Declaration: `FUNCTION msp_CTM5(VAR Var1:integer;VAR Var2;VAR Var3: integer):integer;external;`

.ASM Subroutine:

The following assembly code shows how the driver handles user arguments:

```
msp_CTM5 proc far           ; far call (dword return address)
        push bp             ; save base pointer
        mov bp,sp           ; save stack pointer
        . ; [bp+4] holds offset of Mode
        . ; [bp+6] holds offset of Params
        . ; [bp+8] holds offset of Flag
        . ; Program execution here
        . ;
        . ;
        mov ax,n            ; Return Value for Function In ax register
        pop bp              ;
        ret 6 ; return and pop bp & sp values prior to exit
msp_CTM5 endp
```

Other:

This information is provided for those wishing to create their own drivers:

- msp_CTM5 is declared "PUBLIC" in the .ASM file
- msp_CTM5 is declared external in the calling program
- msp_CTM5 resides in segment_TEXT (default of the .model command)

Microsoft PASCAL Example

```
(*******************************************************)
(* MCPEXAMPL.PAS                                       *)
(* CTM-05 EXAMPLE OF MODE 0                            *)
(* USING MICROSOFT PASCAL                              *)
(*******************************************************)

Type

Parray = array[1..16] of word;

Var

Params    : Parray;
Mode,Flag : integer;
Result    : integer;

(* Define Driver Function Call *)

FUNCTION msp_CTM5(VAR Mode:integer;VAR Params:Parray;VAR Flag:integer):
   INTEGER;EXTERN;

(* MAIN *)

BEGIN
   Mode:= 0;
   Params[1]:= 768;    (* Base Address of Board *)
   Params[2]:= 10;     (* FOUT Divider Ratio of 10 *)
   Params[3]:= 15;     (* Compare 2 Disabled *)
   Params[4]:= 0;      (* Compare 2 Disabled *)
   Params[5]:= 0;      (* Compare 1 Disabled *)
   Params[6]:=0;       (* Time Of Day Disabled *)

     Result:= msp_CTM5(Mode,Params,Flag);
```

```
    if(Result <> 0) then
      WriteLn('Mode 0 Error # = ',Result);

(* REMAINDER OF CODE *)
.
END.
```

2.4 BORLAND TURBO PASCAL (VER 3.0 - 4.0)

Borland's Turbo PASCAL supports a compact and a large memory model. The compact model supports one code segment and multiple data segments. In this model, the code segment is limited to 64K with assembly routine calls being near calls. The data segment is unlimited. The large model permits unlimited code and data segments with assembly calls and data access being far calls.

The program (TINST.EXE) shipped with TURBO PASCAL can change the calling convention so that the user may not know which convention they are using. The default state is "OFF" or compact mode. In order to ascertain which mode you are using, run the "TINST.EXE" program.

Compact Model

Model:	Compact (Forces far call "OFF" in TINST.EXE)
Passes:	dword size pointers (offset and segment)
Sequence:	Arguments Passed Left to Right
Default Call Convention:	Arguments Passed by Value

Example

PASCAL Call: `Result: = tp_CTM5 (Var1, Var2, Var3);`

PASCAL Declaration: `FUNCTION tp_CTM5(VAR Var1:integer;VAR Var2;VAR Var3: integer):integer;external;`

.ASM Subroutine (Either Model):

The following assembly code shows how the driver handles user arguments:

```
tp_CTM5 proc near           ; near call (single word return address)
        push bp             ; save base pointer
        mov bp,sp           ; save stack pointer
        . ; [bp+4] holds offset of VAR3
        . ; [bp+6] holds offset of VAR2
        . ; [bp+8] holds offset of VAR1
        . ; Program execution here
        . ;
        . ;
        mov ax,n            ; return Value for Function In ax register
        pop bp
        ret 12              ; return & pop values prior to exit
tp_CTM5 endp
```

Other:

This information is provided for those wishing to create their own drivers:

- Use the $L 'Metacommand' to link the object file containing external function tp_CTM5, i.e. {$l tpucCTM5} (Link to file tpucCTM5.obj).
- The VAR declarative forces pass by reference (address of variable) in the function declaration. Default is pass by value (pushing the actual integer value onto the stack).
- tp_CTM5 is declared external in the calling program along with the type of return value (integer). Remember that in PASCAL, functions return a value whereas procedures never do.
- The .ASM file contains an explicit declaration of the code segment containing tp_CTM5. Turbo PASCAL handles segments in a primitive manner which is not compatible with the '.model' statements available in MASM or TASM. The function tp_CTM5 must reside in a segment called 'CODE'! Turbo PASCAL will not accept any other segment name. If tp_CTM5 is not in segment "CODE", the linker returns an "unresolved external" error. The Segment Declaration for "CODE" in the .ASM file must appear as:

```
CODE SEGMENT WORD PUBLIC
ASSUME CS:CODE
.
.       ; CODE GOES HERE
.
CODE ENDS
```

Large Model

Model:	Large (Forces far call "ON" in TINST.EXE)
Passes:	dword size pointers (offset and segment)
Sequence:	Arguments Passed Left to Right
Default Call Convention:	Arguments Passed by Value

Example

PASCAL Call: `Result: = tp_CTM5 (Var1, Var2, Var3);`
PASCAL Declaration: `FUNCTION tp_CTM5(VAR Var1:integer;VAR Var2;VAR Var3: integer):integer;external;`

.ASM Subroutine (Either Model):
The following assembly code shows how the driver handles user arguments:

```
tp_CTM5 proc far              ; far call (dword return address)
        push bp               ; save base pointer
        mov bp,sp             ; save stack pointer
        . ; [bp+4] holds dword of VAR3
        . ; [bp+8] holds dword of VAR2
        . ; [bp+12] holds dword of VAR1
        . ; Program execution here
        . ;
        . ;
        mov ax,n              ; return Value for Function In ax register
        pop bp
        ret 12                ; return & pop values prior to exit
tp_CTM5 endp
```

Other:

This information is provided for those wishing to create their own drivers:

- Use the $L 'Metacommand' to link the object file containing external function tp_CTM5. For example; {$l tpucCTM5} (Link file tpucCTM5.obj).
- The VAR declarative forces pass by reference (address of variable) in the function declaration. Default is pass by value (pushing the actual integer value onto the stack).
- tp_CTM5 is declared external in the calling program along with the type of return value (integer). Remember, in PASCAL, functions return a value procedures don't.
- The .ASM file contains an explicit declaration of the code segment containing tp_CTM5.

Turbo PASCAL Example

```
T{S$R-}
{$I-}
{$B+}
{$S+}
{$N-}
{$L T0CTM5}
{$M 65500, 16384, 655360}

(*************************************************************)
(* TPEXAMPLE.PAS                                             *)
(* CTM-05 EXAMPLE OF MODE 0                                  *)
(* USING TURBO PASCAL                                        *)
(*************************************************************)
Type
Parray = array[1..16] of word;
Var

Params    : Parray;
Mode,Flag : integer;
Result    : integer;

(* Define Driver Function Call *)
FUNCTION tp_CTM5(VAR Mode:integer;VAR Params:Parray;VAR Flag:integer):
  INTEGER;EXTERN;

(* MAIN *)
BEGIN
  Mode:= 0;           (* Use Mode 0 *)
  Params[1]:= 768;    (* Base Address of Board *)
  Params[2]:= 10;     (* FOUT Divider Ratio of 10 *)
  Params[3]:= 15;     (* FOUT Source *)
  Params[4]:= 0;      (* Compare 2 Disabled *)
  Params[5]:= 0;      (* Compare 1 Disabled *)
  Params[6]:= 0;      (* Time Of Day Disabled *)
    Result:= tp_CTM5(Mode,Params,Flag);
    if(Result <> 0) then
      WriteLn('Mode 0 Error # = ',Result);

(* REMAINDER OF CODE *)
.
.
END.
```

2.5 MICROSOFT FORTRAN (V4.0 AND UP)

Large Model

Model:	Large
Passes:	dword size pointers (offset and DS register)
Sequence:	Arguments Passed Left to Right
Default Call Convention:	Arguments Passed by Reference

Example

FORTRAN Call:	`call fCTM5(Var1, Var2, Var3);`
FORTRAN Declaration:	None necessary in FORTRAN source file (Fortran assumes that undeclared subroutines or functions are external. It is left to the linking process to provide the required .LIB or .OBJ files. However, the function name should conform to ANSI FORTRAN rules for integer functions.

.ASM Subroutines:

NOTE: FORTRAN integer functions (beginning with letters i, j, or k) return results in the ax register whereas non-integer functions reserve 4 bytes on the calling stack for a far pointer to the result. Non-integer functions pass their arguments starting at location bp+10 after the "push bp" and "mov bp,sp" instructions have been executed. The FORTRAN <--> Assembly routines predominantly use type integer to avoid the non-integer problem. Using non-integer functions may be a problem when returning pointers, floating point results, long integers, etc. The user should use the IMPLICIT INTEGER (A-Z) declaration causing all Functions and Variables to be implicitly type integer unless declared otherwise. Also note that FORTRAN calls by Reference. This method places the address of the passed parameters (rather than the parameters themselves) onto the stack at the time of the call to any function or subroutine. As a convenience, PCF-CTM05 provides two functions (INBYT and OUTBYT) for directly addressing the registers (see example below for syntax and usage).

FORTRAN Example

```
C       INOUT.FOR
C       Example for using INBYT & OUTBYT Functions

        program inout
        integer*2 port,outdat
        integer*1 indat

        port=0
        outdat=0

        do 35 i=1,10,1
        write (*,10)
10      format('Enter Port Address(Decimal): ')

        read(*,15) port
15      format(i3)
```

```
        write(*,20)
      20 format(' Enter data to write(-1 = exit) ')

        read(*,25) outdat
      25 format(i3)

        if(outdat .EQ. -1) go to 45

        write (*,30) outdat
      30 format(' Data Written = ',z)

        call outbyt(port,outdat)
        indat=inbyt(port)

      35 write(*,40) indat
      40 format(' Data Read = ',z)
      45 end
```

Integer (Default) Function or Subroutine

The following assembly code shows how the driver handles user arguments:

```
fCTM5   proc far              ; dword pointer return address
        push bp               ; save base pointer
        mov bp,sp             ; save stack pointer
        . ; [bp+6] holds offset of VAR3
        . ; [bp+10] holds offset of VAR2
        . ; [bp+14] holds offset of VAR1
        . ; Program execution here
        . ;
        . ;
        mov ax,n              ; return Value for Function In ax register
        pop bp
        ret   ;
fCTM5   endp
```

NOTES:

1. VAR3 = Return Value of Function
2. Function fCTM5 must be declared as an integer * 2 function.

Microsoft FORTRAN Example

```
C*************************************************************************
C*      MSFEXAMPLE.FOR
C*      CTM-05 EXAMPLE OF MODE 0
C*      USING MICROSOFT FORTRAN
C*************************************************************************

integer*2 Params(16), Mode, Flag, fCTM5

        Mode = 0;                       (* Use Mode 0 *)
        Params(1):= 768;                (* Base Address of Board *)
        Params(2):= 10;                 (* FOUT Divider Ratio of 10 *)
        Params(3):= 15;                 (* FOUT Source *)
        Params(4):= 0;                  (* Compare 2 Disabled *)
        Params(5):= 0;                  (* Compare 1 Disabled *)
        Params(6):= 0;                  (* Time Of Day Disabled *)
```

```
        call fCTM5(Mode, Params(1), Flag);

          if (Flag .NE. 0) then
          print *,'Mode 0 Error # =',Flag

REMAINDER OF CODE
.
.
```

2.6 INTERPRETED BASIC (GW, COMPAQ, IBM, ETC.)

Medium Model (Only Model Available)

Model:	Medium (Far Calls, Single Data)
Passes:	word size pointers (offset and no DS Register)
Sequence:	Arguments Passed Left to Right
Default Call Convention:	Arguments Passed by Reference

Example

BASIC Call:	`12500 CALL CTM5(MODE%, PARAMS%(0), FLAG%)`
BASIC Declaration:	NONE NECESSARY IN BASIC SOURCE CODE. However, a "BLOAD" (Binary load of .BIN file) of the binary file containing the external subroutine must be done prior to calling that subroutine.

.ASM Subroutine:
The following assembly code shows how the driver handles user arguments:

```
Location 0 (Beginning of Code Segment)

        jmp CTM5

        .
        .
CTM5    proc far              ; far call (dword return address)
        push bp               ; save base pointer
        mov bp,sp             ; save stack pointer
        . ; [bp+6] holds offset of Mode
        . ; [bp+8] holds offset of Params
        . ; [bp+10] holds offset of Flag
        .
        . ; Program execution here
        .
        .
        pop bp                ; restore bp & sp prior to exit
        ret
CTM5    endp
```

NOTE BASIC requires that the .BIN file containing the callable subroutine "CTM5(Mode%, Params%(0), Flag%)" reside at location 0 in the .ASM segment or to "jmp" (unconditional jump) to the .BIN file. A BASIC "jmp " will always jump to location 0 in the .ASM code segment.

Creation of a .BIN file is accomplished as follows:

1. Create the .ASM Source Code File 'EXAMPLE.ASM'

2. Assemble 'EXAMPLE.ASM' thus creating 'EXAMPLE.OBJ'

3. Link 'EXAMPLE.OBJ' to create 'EXAMPLE.EXE'

4. Run EXE2BIN on 'EXAMPLE.EXE' (DOS Utility) to create 'EXAMPLE.COM'

5. Run MAKEBIN.EXE (CTM-05 Software Utility) on 'EXAMPLE.COM' to create

'EXAMPLE.BIN'

MASM EXAMPLE ;

LINK EXAMPLE ;

EXE2BIN EXAMPLE.EXE EXAMPLE.COM

MAKEBIN EXAMPLE.COM

The .BIN file is loaded at a certain location within a specified segment defined by the "DEF SEG" command. This location is then supplied to IBM BASIC via a pointer residing at locations &h510 and &H511. This allows the user to perform a BLOAD at a known address in relation to BASIC's starting address. GW-BASIC does not supply this information so that the user must specify the address when BLOADing the .BIN file. Notice that the example program arbitrarily uses &H8000 for the BLOAD segment. Caution should be exercised, however, to avoid overwriting any existing programs loaded in high memory.

The Following Example Program Illustrates a BASIC CALL:

```
100  '*************************************************************************
110  '* BASEXAMP.BAS
120  '* CTM-05 EXAMPLE OF MODE 0
130  '* USING BASIC
140  '*************************************************************************
150  SG = &H8000
160  DEF SEG = SG
170  BLOAD "CTM5.BIN", 0
180  DIM PARAMS%(15)
190  MODE% = 0                'USE MODE 0
200  PARAMS%(0) = 768         'BASE ADDRESS
210  PARAMS%(1) = 10          'FOUT DIVIDER RATIO OF 10
220  PARAMS%(2) = 15          'FOUT SOURCE
222  PARAMS%(3) = 0           'COMPARE 2 DISABLED
224  PARAMS%(4) = 0           'COMPARE 1 DISABLED
226  PARAMS%(5) = 0           'TIME OF DAY DISABLED
230  CALL CTM5(MODE%,PARAMS%,FLAG%)      'CALL TO DRIVER
240  '
250  IF FLAG <> 0 THEN PRINT "MODE 0 ERROR #",FLAG
260  '
270  .
280  .
     .
     etc.
```

QUICKBASIC

Medium Model (Only Model Available)

Model:	Medium (Far Calls, Single Data)
Passes:	word size pointers (offset and no DS Register)
Sequence:	Arguments Passed Left to Right
Default Call Convention:	Arguments Passed by Reference

Example

BASIC Call: `CALL QBCTM5(MODE%, PARAMS%(0), FLAG%)`

BASIC Declaration: The Declaration tells QuickBASIC that the subroutine expects three arguments and that the middle argument is to be passed by value. Remember that BASIC normally passes all arguments by reference (address). This is the only method for passing an array to a subroutine in BASIC: passing the value of the address of the array in effect passes the array by reference. To make use of the callable assembly routine, a ".QLB" (Quick Library) file is created out of the original .ASM source file. Although the format of the subroutine is identical to those used by interpreted BASIC packages, both the Quick BASIC integrated development environment (QB.EXE) and the command line comelier (BC.EXE) expect the subroutine to be in a specially formatted .QLB library file. Unlike interpreted BASIC packages, Quick BASIC actually links to the assembly .QLB library file so it is not necessary to include the "jmp QBCTM5" instruction at location 0 (of the source file) as in interpreted BASIC.

.ASM Subroutine:
The following assembly code shows how the driver handles user arguments:

```
QBCTM5  proc far              ; far call (dword return address)
        push bp               ; save base pointer
        mov bp,sp             ; save stack pointer
        .  ; [bp+6] holds offset of Mode
        .  ; [bp+8] holds offset of Params
        .  ; [bp+10] holds offset of Flag
        .
        .  ; Program execution here
        .
        .
        pop bp                ; restore bp & sp prior to exit
        ret
QBCTM5  endp
```

NOTE When creating a .QLB file, it is good practice to make a .LIB of the same version as a backup file.

Creation of a .QLB file is accomplished as follows:

1. Create the .ASM Source Code File 'EXAMPLE.ASM'
2. Assemble 'EXAMPLE.ASM' thus creating 'EXAMPLE.OBJ'

3. Link 'EXAMPLE.OBJ' with the "/q" option to create 'EXAMPLE.QLB'

MASM EXAMPLE;

LINK /q EXAMPLE;

A .LIB file is created by:

1. Create the .ASM Source Code File 'EXAMPLE.ASM'
2. Assemble 'EXAMPLE.ASM' thus creating 'EXAMPLE.OBJ'
3. Use Utility LIB.EXE to add EXAMPLE.OBJ to 'EXAMPLE.LIB'

(Remove old EXAMPLE.OBJ from Library)

LIB EXAMPLE.LIB -EXAMPLE

(Create New .OBJ) **MASM EXAMPLE;**

(Add New .OBJ to Library) **LIB EXAMPLE,LIB +EXAMPLE;**

4. To use the .QLB file in the QB integrated environment/editor, invoke QB.EXE with the /l option (QB /l qlbname.qlb,) where qlbname.qlb is the file containing BASICsub.

5. To use the .LIB file with the command line comelier (BC.EXE), simply specify "EXAMPLE.LIB" in the link process.

The Following Example Program Illustrates a QuickBASIC CALL:

```
'*************************************************************************
'* QBEXAMP.BAS
'* CTM-05 EXAMPLE OF MODE 0
'* USING QuickBASIC
'*************************************************************************

DIM PARAMS%(15)
COMMON SHARED PARAMS%()

DECLARE SUB QBCTM5(MODE%, BYVAL DUMMY%, FLAG%)

MODE% = 0    'USE MODE 0
PARAMS%(0) = 768            'BASE ADDRESS OF BOARD
PARAMS%(1) = 10             'FOUT DIVIDER RATIO OF 10
PARAMS%(2) = 15             'FOUT SOURCE
PARAMS%(3) = 0              'COMPARE 2 DISABLED
PARAMS%(4) = 0              'COMPARE 1 DISABLED
PARAMS%(5) = 0              'TIME OF DAY DISABLED

CALL QBCTM5(MODE%,VARPTR(PARAMS%(0)),FLAG%)    'CALL TO DRIVER

IF FLAG <>0 THE PRINT "MODE 0 ERROR #",FLAG

MORE CODE
.
.
```

2.7 CTM5.LIB GENERAL PURPOSE LIBRARY

CTM5.LIB — This is a general purpose library file which provides control of the CTM-05. This file can be linked with programs written in C, PASCAL, FORTRAN, or QuickBASIC to provide access to the CTM-05 operating modes.

NOTE: This library cannot be used with Turbo PASCAL. However, Turbo PASCAL may be used with Turbops.obj (see below).

The following is a brief description of the available call routines:

mscs_ctm5(mode,param,flag)	:	Call from Microsoft C Small Model
mscm_ctm5(mode,param,flag)	:	Call from Microsoft C Medium Model
mscl_ctm5(mode,param,flag)	:	Call from Microsoft C Large Model
tcs_ctm5(mode,param,flag)	:	Call from Turbo C Small Model
tcm_ctm5(mode,param,flag)	:	Call from Turbo C Medium Model
tcl_ctm5(mode,param,flag)	:	Call from Turbo C Large Model
msp_ctm5(mode,param,flag)	:	Call from Microsoft PASCAL
qbctm5(mode,param,flag)	:	Call from Microsoft QuickBASIC
fctm5(mode,param,flag)	:	Call from Microsoft FORTRAN

Linking the Library "CTM5.lib" to the user program is accomplished after program compilation by including it in the link line as follows:

```
link userprog.obj,userprog,,user.lib_ctm5.LIB;
```

userprog.obj is an object module produced by compilation of the user program.
userprog should be used for the resultant executable .EXE file.
user.lib is any other user library, if applicable.

For Turbo PASCAL, the entry point is:

```
tp_ctm5(mode,param,flag)     :     Call from Turbo PASCAL program
```

The user program should have the directive

```
{$L tpctm5} if using the PCF-CTM05
```

■ ■ ■

The following is a listing, by name and category, of the files that should be on PCF-CTM05:

Driver Source Files

FILE NAME	DESCRIPTION
CTM5.ASM	Source code for Driver MODEs 1-11.
CTM5PCF.ASM	Source code of Language Interface Routines.
TPCTM5.ASM	Source code for Turbo PASCAL Interface Routines.

Driver .LIB and .OBJ Files

FILE NAME	DESCRIPTION
CTM5.LIB	Driver for C, PASCAL and FORTRAN.
TPCTM5.OBJ	CTM-05 driver for Turbo PASCAL.

Documentation and Executable Programs:

FILE NAME	DESCRIPTION
FILES.DOC	A list of all the distribution files in this package.
README.DOC	Information about current software version.
HOWTOEXE.DOC	Information on how to create .EXE files from some of the available files.
CTM5.DOC	C, PASCAL, and FORTRAN Language Interface Info file.
ECONFIG.EXE	Used to modify the CONFIG.SYS file.

Source C, PASCAL, and FORTRAN Programs

Refer to FILES.DOC on the distribution diskette(s) for a complete listing of example programs.

Documentation Files

FILE NAME	DESCRIPTION
FILES.DOC	A list of all the distribution files in this package.
CTM5.DOC	C, PASCAL, and FORTRAN language interface info file.
README.DOC	Information about current software version.

■ ■ ■

❑

❑

❑