What is the difference between PIN macros and PIO functions?

The PIOxxxx functions and PinXXXX macros were originally designed to support the 68338 based CF1, on which all of the general purpose I/O pins were under direct control of the CPU through registers for simple read, write, and configuration operations. Service times for these pins are deterministic and near instantaneous depending on the invocation method as described ahead.

The 68332 based CF2 replaced the CTM module with a more capable TPU module, but lost direct control of the TPU I/O pins to an indirect mechanism of service requests and acknowledgement transactions between the CPU and TPU module via internal registers and a shared dual-port ram. Service times for these pins varies with TPU loading, and valid pin responses are only guaranteed for transactions where the CPU waits for completion which is a function of the invocation method as described ahead. The CF2 TPU controlled I/O pins includes the 15 pins named TPU1 through TPU15 and the CPU controlled I/O pins include:

C40	/IRQ7	10k pull-up, must be high at reset
C39	/IRQ5	10k pull-up, must be high at reset, (aka PBM)
C41	/IRQ2	10k pull-up
C48	/RTS	connects to MAX3222 EIA level shifter
C50	/CTS	connects to MAX3222 EIA level shifter
C42	MODCLK	10k pull-up, must be high at reset
<all QSPI pins>		
C1	/DS	drives data strobe out at reset

INVOCATION METHODS (PIOxxxx functions versus PinXXXX macros)

The CF1 and CF2 uses two methods to control the CPU controlled I/O pins, with the Pin inline assembly language macros offering far better raw speed in exchange for sometimes quirky behaviors when compared to the PIO functions. The PIO functions always do what is requested in that they force the pin from its alternate function and force the I/O direction to match the request before actually performing the action. At 16MHz, CPU controlled PIO functions take on the order of 40us. The Pin macros exist because they take only a fraction of the time required by the PIO functions for CPU controlled I/O pins. However, they only work if the pins have previously been properly setup as I/O and specifying direction with at least one earlier PIO request.

So why use Pin macros for TPU pins? It's because they take only several microseconds and the majority of pin operations are not used in tight loops or with synchronization constraints. A good rule of thumb is to use only PIO functions unless you need the performance of the Pin macros and then use them sparingly and with the proper setup.

The CF2 TPU controlled I/O pins retain both the Pin inline assembly language macros and the PIO functions for portability and orthogonality, but at high speeds, the benefits are less and the quirkiness is greater. For TPU PIO functions, the 16MHz requests take on the order of 60us, but that time may increase as the TPU workload increases (e.g. high-speed PWM, multiple fast UARTs). The TPU Pin macros also use inline assembly language for a speed boost, but the big speed gains come from ignoring the completion status which requires a loop and accounts for the bulk of normal TPU operation request. This becomes an issue when multiple TPU operations will be requested without an intervening 50us to allow the previous operation to complete. The TPU I/O pins simply cannot keep up with a barrage of Pin macro commands because they do not wait for acknowledgement before returning control to the program. This is most evident by putting a PinToggle request in a tight loop - the pin will just stay in the same state because new commands come in before the last one can complete. Inserting a 50us delay in the loop will remedy the problem. Of note, the same loop with a CPU pin and no delay will generate a 120kHz square wave.

The code snippet below shows three TPU GPIO access methods with cycle times of: PIO 110us, Pin with sys call 34us, and Pin with extra macro 16us. Variations on these may help in translating CF1 code to the CF2.

```
bool    M_TPUHostServiceCheckComplete(ushort tch:__d0):__d0 = \
            { 0xD080, 0x7203, 0xE1A9, 0xC2B8, 0xFE18, 0x57C1, 0x2001 };

void main(void)
    {
    ushort  test_tch = TPUChanFromPin(25);

    // Identify the progam and build
    printf("\nProgram: %s: %s %s \n", __FILE__, __DATE__, __TIME__);
    // Identify the device and its firmware
    printf("Persistor CF%d SN:%ld   BIOS:%d.%02d   PicoDOS:%d.%02d\n", CFX,
        BIOSGVT.CFxSerNum, BIOSGVT.BIOSVersion, BIOSGVT.BIOSRelease,
        BIOSGVT.PICOVersion, BIOSGVT.PICORelease);

    while (! kbhit())       // 110us, 9kHz
        {
        PIOSet(25);         // 50us HT
        PIOClear(25);       // 60us LT
        }
    cgetc();    // clear


    while (! kbhit())       // 34us, 30kHz
        {
        PinSet(25);         // 12us HT
        TPUHostServiceCheckComplete(25, true);
        PinClear(25);       // 22us LT
        TPUHostServiceCheckComplete(25, true);
        }
    cgetc();    // clear


    while (! kbhit())       // 16us, 61kHz
        {
        PinSet(25);         // 3us HT
        while(! M_TPUHostServiceCheckComplete(test_tch))
            ;
        PinClear(25);       // 13us LT
        while(! M_TPUHostServiceCheckComplete(test_tch))
            ;
        }
    cgetc();    // clear

    }   //____ main() ____//
```