Object Oriented Extensions

These words provide a fast and compact object-oriented capability to MaxForth. It defines Forth words as "methods" which are associated only with objects of a specific class.

Action of an Object

An object is very much like a <BUILDS DOES> defined word. It has a user-defined data structure which may involve both Program ROM and Data RAM. When it is executed, it makes the address of that structure available (though not on the stack...more on this in a moment).

What makes an object different is that there is a "hidden" list of Forth words which can only be used by that object (and by other objects of the same class). These are the "methods," and they are stored in a private wordlist. *Note that this is not the same as a Forth "vocabulary." Vocabularies are not used, and the programmer never has to worry about word lists.*

Each method will typically make several references to an object, and may call other methods for that object. If the object's address were kept on the stack, this would place a large burden of stack management on the programmer. To make object programming simpler *and* faster, the address of the current object is stored in a variable, `OBJREF`. The contents of this variable (the address of the current object) can always be obtained with the word `SELF`.

When *executed (interpreted)*, an object does the following:
1. Make the "hidden" word list of the object available for searching.
2. Store the object's address into OBJREF.

After this, the private methods of the object can be executed. (These will remain available until an object of a different class is executed.)

When *compiled*, an object does the following:
1. Make the "hidden" word list of the object available for searching.
2. Compile code into the current definition which will store the object's address into OBJREF.

After this, the private methods of the object can be compiled. (These will remain available until an object of a different class is compiled.) *Note that both the object address and the method are resolved at compile time. This is "early binding" and results in code that is as fast as normal Forth code.*

In either case, the syntax is identical:
```
       object method
```
For example:
```
       REDLED TOGGLE
```

Defining a new class

`BEGIN-CLASS name`

Words defined here will only be visible to objects of this class.
These will normally be the "methods" which act upon objects of this class.

`PUBLIC`

Words defined here will be visible at all times.
These will normally be the "objects" which are used in the main program.

`END-CLASS name`

Defining an object

`OBJECT name` This defines a Forth word "name" which will be an object of the current class. The object will initially be "empty", that is, it will have no ROM or RAM allocated to it. The programmer can add data structure to the object using `P,` `,` `PALLOT` and `ALLOT`, in the same manner as for <BUILDS DOES> words. *Like <BUILDS DOES>, the action of an object is to leave its **Program** memory address.*

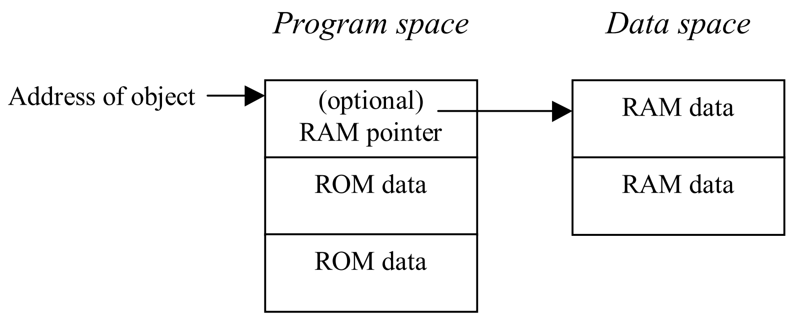
Referencing an object

`SELF` This will return the address of the object last executed. *Note that this is an address in **Program** memory. If the object will use Data RAM, it is the responsibility of the programmer to store a pointer to that RAM space. See the example below.*

Object Structure

An object may have associated data in both Program and Data spaces. This allows ROM parameters which specify the object (e.g., port numbers for an I/O object); and private variables ("instance variables") which are associated with the object. By default, objects return their Program (ROM) address. If there are RAM variables associated with the object, a pointer to those variables must be included in the ROM data.

Object data structure



Note that also `OBJECT` creates a pointer to Program space, it does not reserve *any* Program or Data memory. That is the responsibility of the programmer. This is done in the same manner as the <BUILDS clause of a <BUILDS DOES> definition, using `P,` or `PALLOT` to add cells to Program space and `,` or `ALLOT` to add cells to Data space. The programmer can use `OBJECT` to build a custom defining word for each class. See the example below.

Example using ROM and RAM

This is an example of an object which has both ROM data (a port address) and RAM data (a timebase value).

```
BEGIN-CLASS TIMERS
  : TIMER ( a -- )  OBJECT  HERE 1 ALLOT P,  P, ;
PUBLIC
  0D00 TIMER TA0
  0D08 TIMER TA1
END-CLASS TIMERS
```

The word `TIMER` expects a port address on the stack. It builds a new (empty) `OBJECT`. Then it reserves one cell of Data RAM (`1 ALLOT`) and stores the starting address of that RAM (`HERE`) into Program memory (`P,`). This builds the RAM pointer as shown above. Finally, it stores the I/O port address "a" into the second cell of Program memory (the second `P,`). *Each* object built with `TIMER` will have its own copy of this data structure.

After the object is executed, `SELF` will return the address of the Program data for that object. Because we've stored a RAM pointer as the first Program cell, the phrase `SELF P@` will return the address of the RAM data for the object. *It is not required that the first Program cell be the RAM pointer, but this is strongly recommended as a programming convention for all objects using RAM storage.*

Likewise, `SELF CELL+ P@` will return the I/O port address associated with this object (since that was stored in the second cell of Program memory by `TIMER`).

We can simplify programming by making these phrases into Forth words. We can also build them into other Forth words. All of this will normally go in the "private" class dictionary:

```
BEGIN-CLASS TIMERS
  : TIMER      ( a -- )  OBJECT  HERE 1 ALLOT P,  P, ;

  : TMR_PERIOD ( -- a )  SELF P@ ;    ( RAM variable for this timer)
  : BASEADDR   ( -- a )  SELF CELL+ P@ ;  ( I/O addr for this timer)
  : TMR_SCR    ( -- a )  BASEADDR 7 + ;   ( Control register )

  : SET-PERIOD ( n -- )  TMR_PERIOD ! ;
  : ACTIVE-HIGH ( -- )   0202 TMR_SCR CLEAR-BITS ;
PUBLIC
  0D00 TIMER TA0      ( Timer with I/O address 0D00 )
  0D08 TIMER TA1      ( Timer with I/O address 0D08 )
END-CLASS TIMERS
```

After this, the phrase `100 TA0 SET-PERIOD` will store the RAM variable for timer object TA0, and `200 TA1 SET-PERIOD` will store the RAM variable for timer object TA1. `TA0 ACTIVE-HIGH` will clear bits in timer A0 (at port address 0D07), and `TA1 ACTIVE-HIGH` will clear bits in timer A1 (at port address 0D0F).

In a `WORDS` listing, only `TA0` and `TA1` will be visible. But after executing `TA0` or `TA1`, all of the words in the `TIMERS` class will be found in a dictionary search.

Because the "methods" are stored in private word lists, you can re-use method names in different classes. For example, it is possible to have an `ON` method for timers, a different `ON` method for GPIO pins, a third `ON` method for PWM pins, and so on. When the object is named, it will automatically select the correct set of methods to be used! Also, if a particular method has *not* been defined for a given object, you will get an error message if you attempt to use that method with that object. (One caution: if there is word in the Forth dictionary with the same name, and there is no method of that name, the Forth word will be found instead. An example of this is `TOGGLE`. If you have a `TOGGLE` method, that will be compiled. But if you use an object that doesn't have a `TOGGLE` method, Forth's `TOGGLE` will be compiled. *For this reason, methods should* ***not*** *use the same names as "ordinary" Forth words.*)

Because the "objects" are in the main Forth dictionary, they must all have unique names. For example, you can't have a Timer named `A0` and a GPIO pin named `A0`. You must give them unique names like `TA0` and `PA0`.

GPIO Bit I/O Class

These words support the GPIO I/O of the DSP56F80x. The following GPIO pins are defined as objects:

```
PA7   PA6   PA5   PA4   PA3   PA2   PA1   PA0
PB7   PB6   PB5   PB4   PB3   PB2   PB1   PB0
PD3   PD2   PD1   PD0
REDLED  YELLED  GRNLED
```

For each pin, the following methods can be performed:

`ON`	Makes the pin an output, and outputs a '1' (high level).
`OFF`	Makes the pin an output, and outputs a '0' (low level).
`TOGGLE`	Makes the pin an output, and inverts its level.
`n SET`	Stores a T/F value to the pin, e.g., `1 PA0 SET`. Any nonzero value is "true."
`GETBIT`	Makes the pin an input, and returns pin value (as a bit mask).
`ON?`	Makes the pin an input, and returns true if pin is '1' (high level).
`OFF?`	Makes the pin an input, and returns true if pin is '0' (low level).
`IS-INPUT`	Makes pin an input (hi-Z).
`IS-OUTPUT`	Makes pin an output. Pin will output the last programmed level.

Examples of use:

```
PA0 OFF      ( output a low level on PA0 )
0 PA0 SET    ( also outputs a low level on PA0 )
REDLED ON    ( output a high level, turn the red LED on )
PD3 ON?      ( check if PD3 is a logic '1' )
```

GPIO Byte I/O Class

These words support the GPIO I/O of the DSP56F80x as bytes. The following GPIO ports are defined as objects:

```
PORTA   PORTB
```

For each pin, the following methods can be performed:

`IS-INPUT`	Makes port an input (hi-Z).
`IS-OUTPUT`	Makes port an output. Pin will output the last programmed level.
`PUTBYTE`	Makes port an output, and outputs the given byte (8 bits).
`GETBYTE`	Makes port an input, and reads it as a byte (8 bits).

Examples of use:

```
55 PORTA PUTBYTE  ( output 55 to GPIO Port A )
PORTB GETBYTE .   ( read GPIO Port B and type its numeric value )
```

Timer I/O Class

These words support the Counter/Timers of the DSP56F80x. The following timers are defined as objects:

```
TA0   TA1   TA2   TA3
TB0   TB1   TB2   TB3
TC0   TC1   TC2   TC3
TD0   TD1   TD2
```

For each Counter/Timer, the following methods can be performed:

`ON`	Makes the counter/timer pin an output, and outputs a '1' (high level).
`OFF`	Makes the counter/timer pin an output, and outputs a '0' (low level).
`TOGGLE`	Makes the counter/timer pin an output, and inverts its level.
`n SET`	Stores a T/F value to the pin, e.g., `1 TA0 SET`. Any nonzero value is "true."
`GETBIT`	Makes the counter/timer pin an input, and returns pin value (as a bit mask).
`ON?`	Makes the counter/timer pin an input, and returns true if pin is '1' (high level).
`OFF?`	Makes the counter/timer pin an input, and returns true if pin is '0' (low level).

The following methods can be used to generate PWM signals and to measure pulse width:

`ACTIVE-HIGH`	Makes the pin "active high" for PWM output or input. For output, `PWM-OUT` will control the *high* pulse width. For input, `PWM-IN` will measure the width of the *high* pulse. The reset default is `ACTIVE-HIGH`.
`ACTIVE-LOW`	Makes the pin "active low" for PWM output or input. For output, `PWM-OUT` will control the *low* pulse width. For input, `PWM-IN` will measure the width of the *low* pulse.
`n PWM-PERIOD`	Specifies the period (frequency) of the PWM output. Values from 100 to FFFF hex are valid. The counter frequency is 2.5 MHz; FFFF hex corresponds to a period of 26.214 msec (38 Hz). `PWM-PERIOD` must be specified before using `PWM-OUT`.
`n PWM-OUT`	Makes the counter/timer pin an output, and outputs a continuous PWM signal with the given duty cycle. Values from 0 to FFFF hex are valid. 0 is a duty cycle of 0% (always off); FFFF is a duty cycle of 100% (always on). 8000 hex gives a duty cycle of 50%. `PWM-PERIOD` must be specified before using `PWM-OUT`.
`PWM-IN`	Makes the counter/timer pin an input, and measures the width of one pulse on that input. Returns a value from 1 to FFFF hex. The counter rate is 2.5 MHz, thus each count is 0.4 usec, and a returned value of 10000 decimal corresponds to 4 msec.

Examples of use:

```
TC0 ON      ( output a high level on the TC0 pin )
TA3 ON?     ( check if TA3 pin, HOME0, is a logic '1' )

DECIMAL 50000 TC1 PWM-PERIOD  ( specify 20 msec period = 50 Hz )
TC1 ACTIVE-HIGH               ( specify active-high output )
HEX 4000 TC1 PWM-OUT          ( output 25% high, 75% low )
```

PWM I/O Class

These words support the PWM generators of the DSP56F80x. The following PWM outputs are defined as objects:

```
PWMA0   PWMA1   PWMA2   PWMA3   PWMA4   PWMA5
PWMB0   PWMB1   PWMB2   PWMB3   PWMB4   PWMB5
```

For each PWM output, the following methods can be performed:

`ON`	Outputs a '1' (high level).
`OFF`	Outputs a '0' (low level).
`TOGGLE`	Inverts the output level.
`n SET`	Stores a T/F value to the pin, e.g., `1 PWMA0 SET`. Any nonzero value is "true."

The following methods can be used to generate PWM signals:

`n PWM-PERIOD`	Initializes the PWM output, and specifies its period (frequency). Values from 100 to 7FFF hex are valid. The effective counter frequency is 2.5 MHz; 7FFF hex corresponds to a period of 13.106 msec (76 Hz). `PWM-PERIOD` must be specified before using `PWM-OUT`. ***Note:** setting the period for **any** "A" PWM will affect all six "A" PWMs. Setting the period for **any** "B" PWM will affect all six "B" PWMs.*
`n PWM-OUT`	Outputs a continuous PWM signal with the given duty cycle. Values from 0 to FFFF hex are valid. 0 is a duty cycle of 0% (always off); FFFF is a duty cycle of 100% (always on). 8000 hex gives a duty cycle of 50%. `PWM-PERIOD` must be specified before using `PWM-OUT`.

The following PWM inputs are defined as objects:

```
FAULTA0   FAULTA1   FAULTA2   FAULTA3   ISA0      ISA1      ISA2
FAULTB0   FAULTB1   FAULTB2   FAULTB3   ISB0      ISB1      ISB2
```

For each PWM input, the following methods can be performed:

`GETBIT`	Returns pin value (as a bit mask).
`ON?`	Returns true if pin is '1' (high level).
`OFF?`	Returns true if pin is '0' (low level).

Examples of use:

```
PWMB0 ON     ( output a high level on the PWMB0 pin )
ISA1 ON?     ( check if ISA1 pin is a logic '1' )

DECIMAL 25000 PWMA1 PWM-PERIOD      ( specify 10 msec period = 100 Hz )
HEX 4000 PWMA1 PWM-OUT        ( output 25% high, 75% low )
```

SPI I/O Class

These words support the SPI port of the DSP56F80x. Only one SPI port is present; it is referenced as object

```
SPI0
```

The following methods can be performed for the SPI port:

`MASTER`	Specifies that the DSP56F80x will act as an SPI Master.
`n BITS`	Specifies the number of bits to be sent by `TX-SPI` and read by `RX-SPI`. Values from 2 to 16 are valid.
`MSB-FIRST`	Specifies that words should be sent and received MSB first.
`LSB-FIRST`	Specifies that words should be sent and received LSB first.
`n MBAUD`	Specifies the bit rate to be used for the SPI port. Four values can be specified: 20 (20 Mbits/sec), 5 (5 Mbits/sec), 2 (2.5 Mbits/sec), and 1 (1.25 Mbits/sec). All other values will be ignored and will leave the baud rate unchanged.
`n TX-SPI`	Transmits one word on the SPI port. This will output 2 to 16 bits on the MOSI pin (Master mode) and generate 16 clocks on the SCLK pin. *This will simultaneously input 2 to 16 bits on the MISO pin (Master mode).*
`RX-SPI`	Receives one word from the SPI port. This word must already have been shifted into the receive shift register; if it has not, RX-SPI will wait for it to be shifted in. *In Master mode, data will only be shifted in when a word is transmitted by TX-SPI. In this mode you should use RX-SPI immediately after TX-SPI to read the data that was received.*

It is acceptable to specify all the SPI parameters after selecting the SPI port. Example of use:

```
SPI0 MASTER 16 BITS MSB-FIRST 5 MBAUD
SPI0 TX-SPI SPI0 RX-SPI
```

The default polarity for the SPI port is CPHA=0, CPOL=1. This means that the SCLK line will be high between words, and that the slave should clock data on the falling edge. (Refer to figure 13-4 in the Motorola DSP56F801-7 Users Manual.)

ADC I/O Class

These words support the A/D converter of the DSP56F80x. The following ADC inputs are defined as objects:

```
ADC0   ADC1   ADC2   ADC3   ADC4   ADC5   ADC6   ADC7
```

Only one method can be used with A/D inputs:

`ANALOGIN`	Reads the A/D input and returns its value. The result is in the range 0-7FF8. (The 12-bit A/D result is left-shifted 3 places.) 7FF8 corresponds to an input of Vref. 0 corresponds to an input of 0 volts.

Example of use:

```
ADC7 ANALOGIN      ( read A/D channel 7, pin AN7 )
```