1. ADVANCED PROGRAMMING

1.1. IsoPod, MinPod, TinyPod, PlugaPod Memory Map

DATA MEMORY

0000 0292	Data RAM (Kernel)
0293 07FF	Data RAM (User)
0800 0BFF	reserved
0C00 0FFF	peripherals
1000 17FF	Data Flash (SAVE- RAM)
1800 1FFF	Data Flash (User)

PROGRAM MEMORY

0000 53FF	Program Flash (Kernel)
5400 7CFF	Program Flash (User)
7D00 7DFF	User Interrupt Vectors
7E00 7FDF	Program RAM (User)
7FE0 7FFF	Program RAM (Kernel*)

* Program RAM is used by the kernel only for the Flash programming routines. This space is otherwise available for the user.

1.2. ServoPod Memory Map

DATA MEMORY

0000 0292	Data RAM (Kernel)
0293 0FFF	Data RAM (User)
1000 17FF	peripherals
1800 1FFF	reserved
2000 2FFF	Data Flash (SAVE- RAM)
3000 3FFF	Data Flash (User)

PROGRAM MEMORY

0000 55FF	Program Flash (Kernel)
5600 7CFF	Program Flash (User)
7D00 7DFF	User Interrupt Vectors
8000 EFFF	Program Flash (User)
F000 F7DF	Program RAM (User)
F7E0 F7FF	Program RAM (Kernel*)

* Program RAM is used by the kernel only for the Flash programming routines. This space is otherwise available for the user.

1.3. Earlier IsoMax Kernels Memory Maps

1.3.1. IsoMax v0.3 Memory Map

DATA MEMORY

0000 04E6	Data RAM (Kernel)
04E7 07FF	Data RAM (User)
0800 0BFF	reserved
0C00 0FFF	peripherals
1000 1BFF	Data Flash (Kernel)
1C00 1FFF	Data Flash (User)

PROGRAM MEMORY

0000 31FF	Program Flash (Kernel)
3200 7DFF	Program Flash (User)
7E00 7FDF	Program RAM (User)
7FE0 7FFF	Program RAM (Kernel*)

* Program RAM is used by the kernel only for the Flash programming routines. This space is otherwise available for the user.

1.3.2. IsoMax v0.6 Memory Map

DATA MEMORY

0000 0245	Data RAM (Kernel)
0246 07FF	Data RAM (User)
0800 0BFF	reserved
0C00 0FFF	peripherals
1000 17FF	Data Flash (SAVE- RAM)
1800 1FFF	Data Flash (User)

PROGRAM MEMORY

0000 13FF	Program Flash (Core)
1400 1FFF	Program Flash (User)
2000 3FFF	Program Flash (User) *'803 and '805 only*
4000 7DFF	Program Flash (Kernel)
7E00 7FDF	Program RAM (User)
7FE0 7FFF	Program RAM (Kernel*)

* Program RAM is used by the kernel only for the Flash programming routines. This space is otherwise available for the user.

1.3.3. IsoMax v0.6 Memory Map – DSP56807

DATA MEMORY

0000 0245	Data RAM (Kernel)
0246 0FFF	Data RAM (User)
1000 17FF	peripherals
1800 1FFF	reserved
2000 2FFF	Data Flash (SAVE- RAM)
3000 3FFF	Data Flash (User)

PROGRAM MEMORY

0000 13FF	Program Flash (Core)
1400 3FFF	Program Flash (User)
4000 7DFF	Program Flash (Kernel)
8000 EFFF	Program Flash (User)
F000 F7DF	Program RAM (User)
F7E0 F7FF	Program RAM (Kernel*)

* Program RAM is used by the kernel only for the Flash programming routines. This space is otherwise available for the user.

1.4. Starting IsoMax State Machines

When the IsoPod is reset, it disables all running state machines. You must explicitly start your state machines as part of your application -- usually, in your autostart code. There are two ways to do this: with INSTALL, or with SCHEDULE-RUNS.

1.4.1. Using INSTALL to start a State Machine

From IsoMax version 0.36 onward, the preferred method of starting state machines is with INSTALL. After you have defined a state machine, you can start it by typing

```
state-name SET-STATE
INSTALL machine-name
```

Note that you must use SET-STATE to specify the starting state of the machine **first.** This is because INSTALL will start the machine immediately. To start more machines, simply INSTALL them one at a time:

```
state-name-2 SET-STATE
INSTALL machine-name-2
state-name-3 SET-STATE
INSTALL machine-name-3
etc.
```

Normally,[1] the state machine will start running immediately at the default rate of 100 Hz. SET-STATE and INSTALL can be used even while other state machines are running, that is, INSTALL will *add* a state machine to an already-running list of state machines.

At present, up to 16 state machines can be INSTALLed. Attempting to INSTALL more than 16 machines will result in the message "Too many machines." To install more machines, you can use UNINSTALL or define a MACHINE-CHAIN (both described below).

SET-STATE and INSTALL can be used interactively from the command interpreter, or as part of a word definition.

1.4.2. Removing a State Machine

INSTALL builds a list of state machines which are run by IsoMax. UNINSTALL will remove the last-added machine from this list. You can use UNINSTALL repeatedly to remove more machines from the list, in a last-in first-out order. For example:

---

[1] The commands COLD, SCRUB, and STOP-TIMER will halt IsoMax. The command SCHEDULE-RUNS will override the INSTALLed state machines and dedicate IsoMax to running a particular machine chain.

```
INSTALL machine-name-1
      ( SET-STATE commands have been omitted for clarity)
INSTALL machine-name-2
INSTALL machine-name-3
     . . .
UNINSTALL            ...removes machine-name-3
UNINSTALL            ...removes machine-name-2
UNINSTALL            ...removes machine-name-1
UNINSTALL            ...removes nothing
```

If there are no state machines running, `UNINSTALL` will simply print the message "No machines."

To remove *all* the `INSTALL`ed state machines with a single command, use `NO-MACHINES`.

1.4.3. Changing the IsoMax Speed

When the IsoPod is reset, IsoMax returns to its default rate of 100 Hz -- that is, all the state machines are performed once every 10 milliseconds. You can change this rate with `PERIOD`. The command

```
n PERIOD
```

will set the IsoMax period to "`n`" cycles of a 5 MHz clock. Thus,

```
DECIMAL 5000 PERIOD ...will execute state machines once per millisecond

DECIMAL 1000 PERIOD ...will execute state machines every 200 microsec.
```

...and so on. You can specify a period from 10 to 65535.[2] (Be sure to specify the `DECIMAL` base when entering large numbers, or you may get the wrong value.) The default period is 50000.

1.4.4. Stopping and Restarting IsoMax

Certain commands will halt IsoMax processing:

the `COLD` command
the `SCRUB` command

[2] Note, however, that very few state machines will be able to run in 2 microseconds (corresponding to `10 PERIOD`). If you specify too small a `PERIOD`, no harm will be done, but IsoMax will "skip" periods as needed to process the state machines.

This is necessary because either COLD or SCRUB can remove state machines from the IsoPod memory.[3] You can also halt IsoMax manually with the command STOP-TIMER.

In all these cases, the timer that runs IsoMax is halted. So, even if you INSTALL new state machines, they won't run. To restart IsoMax you should use the command ISOMAX-START. This command will

a) Remove all installed state machines, and
b) Start IsoMax at the default rate of 100 Hz.

Since ISOMAX-START removes all installed state machines, you must use it *before* you use INSTALL. For example:

```
STOP-TIMER
     . . .
ISOMAX-START
state-name-1 SET-STATE
INSTALL machine-name-1
state-name-2 SET-STATE
INSTALL machine-name-2
state-name-3 SET-STATE
INSTALL machine-name-3
```

Resetting the IsoPod does the same as ISOMAX-START: it will remove all installed state machines, and reset the timer to the default rate of 100 Hz.

1.4.5. Running More Than 16 Machines

INSTALL can install both state machines and *machine chains*. A "machine chain" is a group of state machines that is executed together. Machine chains, like state machines, are compiled as part of the program:

```
MACHINE-CHAIN chain-name
     machine-name-1
     machine-name-2
     machine-name-3
END-MACHINE-CHAIN
```

This example defines a chain with the given name, and includes the three specified state machines (which must already have been defined). A machine chain can include any number of state machines.

You must still set the starting state for each of the state machines in a machine chain, before you install the chain. So, you could start this example chain with:

---

[3] The command FORGET can also remove state machines from memory. Be very careful when using FORGET that you don't remove an active state machine; or use STOP-TIMER to halt IsoMax first.

```
state-name-1 SET-STATE          ...a state in machine-name-1
state-name-2 SET-STATE          ...a state in machine-name-2
state-name-3 SET-STATE          ...a state in machine-name-3
INSTALL chain-name
```

You can of course UNINSTALL a machine chain, which will stop all of its state machines.

1.4.6. Using SCHEDULE-RUNS

Prior to IsoMax version 0.36, the preferred method of starting state machines was with SCHEDULE-RUNS.[4] SCHEDULE-RUNS worked only with machine chains, and required you to specify the IsoMax period when you started the machines:

```
EVERY n CYCLES SCHEDULE-RUNS chain-name
```

SCHEDULE-RUNS is still available in IsoMax, to allow older IsoMax programs to be compiled. **However,** you should be aware that using SCHEDULE-RUNS will *disable* any machines started with INSTALL. SCHEDULE-RUNS *replaces* any previously running state machines -- including any previous use of SCHEDULE-RUNS -- and there is no "uninstall" function for it. After using SCHEDULE-RUNS, the only ways to "reactivate" the INSTALL function are

a) use the ISOMAX-START command, or
b) reset the IsoPod

ISOMAX-START will disable any machine chain started by SCHEDULE-RUNS, and will re-initialize IsoMax. You can then INSTALL state machines as described above.

You can use the PERIOD command to change the speed of a machine chain started with SCHEDULE-RUNS.

1.4.7. Autostarting State Machines

When the IsoPod is reset, all state machines are halted. (Strictly speaking, the IsoMax timer is running, but the list of installed state machines is empty.) To automatically start your state machines after a reset, you must write an autostart routine, which uses SET-STATE and INSTALL to start your machines. For example:

```
: MAIN
     state-name-1 SET-STATE
     INSTALL machine-name-1
```

[4] Some versions of IsoMax prior to version 0.36 have a different implementation of INSTALL. That implementation does not work as described here, so for those versions of IsoMax we recommend you use SCHEDULE-RUNS.

```
        state-name-2 SET-STATE
        INSTALL machine-name-2
        state-name-3 SET-STATE
        INSTALL machine-name-3

        . . . more startup code . . .
        . . . application code . . .

    ; EEWORD

    SAVE-RAM
    HEX 7C00 AUTOSTART MAIN
```

In this example, the word `MAIN` is executed when the IsoPod is reset. The first thing it does is to install three state machines. Note that these machines will begin running immediately. If you need to do some initialization before starting these machines, that code should appear before the first `INSTALL` command.

Refer to "Autostarting an IsoMax Application" for details about using `SAVE-RAM` and `AUTOSTART`.

1.5. IsoMax State Machine Language Reference

This illustrates the different options for defining state machines, states, and state transitions.

1.5.1. Defining State Machines

A state machine is defined by name:

```
MACHINE <name-of-machine>
```

If the machine will be moved to Flash ROM, the MACHINE declaration must be immediately followed by EEWORD:

```
MACHINE <name-of-machine> EEWORD
```

1.5.2. Declaring States

Once a state machine has been defined, all of the states which will be part of that machine must be named:

```
ON-MACHINE <name-of-machine>
     APPEND-STATE <name-of-new-state>
     APPEND-STATE <name-of-new-state>
     ...
     APPEND-STATE <name-of-new-state> WITH-VALUE <n> AT-ADDRESS <a> AS-TAG
```

The last example above illustrates a debugging option which is available for states. If WITH-VALUE ... AT-ADDRESS are specified, the value 'n' will be stored at address 'a' when a transition is made *to* this state.[5]

If the state machine will be moved to Flash ROM, *each* state declaration must be immediately followed by EEWORD, thus:

```
ON-MACHINE <name-of-machine>
     APPEND-STATE <name-of-new-state> EEWORD
     APPEND-STATE <name-of-new-state> EEWORD
     ...
     APPEND-STATE <name-of-new-state> WITH-VALUE <n> AT-ADDRESS <a> AS-TAG
       EEWORD
```

1.5.3. Defining States

After the states have been named, the transitions between the states can be defined:

[5] This value is actually stored by either `TO-HAPPEN`, `THIS-TIME`, or `NEXT-TIME`, when they are used by another state to select this as the new state. The tag value is not stored when `SET-STATE` is used.

```
IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> TO-HAPPEN

IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> THIS-TIME

IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> NEXT-TIME
```

<boolean computation> must be a fragment of procedural (Forth) code which leaves a true/false (nonzero/zero) condition on the stack. If the result of this computation is true, the actions following CAUSES (a compound action and a new state) will be performed.

<compound action> is an optional fragment of procedural (Forth) code which is performed when the transition condition is satisfied, and before the state transition actually takes place. This must be stack-neutral (the completed action must take nothing from, and leave nothing on, the stack).

Usually when a transition is made to a new state, that state will be evaluated -- that is, all of its CONDITION clauses will be examined -- on the *next* IsoMax cycle. This is the safest approach, and ensures that all state machines receive adequate service. This is what happens when you specify the next state TO-HAPPEN or NEXT-TIME. (TO-HAPPEN is a synonym for NEXT-TIME).

There may be very special cases when it is important to evaluate the new state *immediately* upon a transition to that state. To achieve this you specify the next-state-name THIS-TIME. This is a hazardous practice, however, since it's very easy to construct a loop of states that will never terminate. THIS-TIME is strongly discouraged, and should only be used when absolutely necessary, and with great care.

If the state machine will be moved to Flash ROM, *each* transition definition must be immediately followed by **IN-EE** (*not* EEWORD), thus:

```
IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> TO-HAPPEN IN-EE

IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> THIS-TIME IN-EE

IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> NEXT-TIME IN-EE
```

1.5.4. Defining Input Conditions

Often the boolean condition in a state transition will simply involve testing an input pin, an I/O register, or a memory location for a bit to be set or cleared. To make this programming easier, you can define an *input trinary*:

```
DEFINE <name>  TEST-MASK <n>  DATA-MASK <m>  AT-ADDRESS <a>  FOR-INPUT
```

The trinary must be given a name. This name acts like a subroutine: when it is used, the trinary tests the value at the specified address, and returns a true/false result. To be precise: the value at address "a" is fetched, and logically ANDed with the `TEST-MASK`. This result is logically XORed with the `DATA-MASK`. If the result is nonzero, a true flag is left on the stack; if the result is zero, a false flag is left.

You should think of this as follows: the `TEST-MASK` specifies which bit is of interest. `DATA-MASK` specifies an optional inversion. Although these will usually act on a single bit, you can certainly have masks with multiple bits. Just remember that if *any* bit in the AND/XOR result is nonzero, the result will be logically "true."

`TEST-MASK`, `DATA-MASK`, and `AT-ADDRESS` can be specified in any order. So, the following are equivalent:

```
DEFINE <name>  TEST-MASK <n>  DATA-MASK <m>  AT-ADDRESS <a>  FOR-INPUT
DEFINE <name>  AT-ADDRESS <a> TEST-MASK <n>  DATA-MASK <m>   FOR-INPUT
DEFINE <name>  DATA-MASK <m>  TEST-MASK <n>  AT-ADDRESS <a>  FOR-INPUT
```

Input trinaries can be used in state transitions and in procedural (Forth) code. **You are not required to use trinaries for the boolean computation in a state transition.** They are merely provided as a convenience.

An input trinary can be moved to Flash ROM with EEWORD:

```
DEFINE <name>  TEST-MASK <n>  DATA-MASK <m>  AT-ADDRESS <a>  FOR-INPUT
EEWORD
```

1.5.5. Defining Output Actions

Many actions involve setting or clearing a bit in an I/O register or a memory location. To make this programming easier, you can define an *output trinary* in one of two forms:

```
DEFINE <name>  SET-MASK <n>  CLR-MASK <m>  AT-ADDRESS <a> FOR-OUTPUT
DEFINE <name>  AND-MASK <n>  XOR-MASK <m>  AT-ADDRESS <a> FOR-OUTPUT
```

The trinary must be given a name. This name acts like a subroutine: when it is used, the trinary sets and clears bits at the specified address.

The most commonly used output action is SET/CLR. When performed, any "1" bits in the `SET-MASK` will be set at address a. Any "1" bits in the `CLR-MASK` will be *cleared* at address a. You can think of this as lists of bits to be set and bits to be cleared in the

register (or memory location). If you need to only set or only clear bits, the unneeeded mask should be zero. For example, to set the LSB at address $F00, you would use

```
DEFINE <name>  SET-MASK 1  CLR-MASK 0  AT-ADDRESS HEX 0F00 FOR-OUTPUT
```

Avoid having the same bit in both the SET-MASK and the CLR-MASK; the result will be indeterminate.[6]

An alternative action is AND/XOR. This can be used to change the state of bits, depending on their current value. When performed, the AND-MASK is applied to the value at address a. Then the XOR-MASK is applied to this result. The final result is stored back to address a. You can thus set, clear, and toggle bits in one operation:

AND-MASK bit	XOR-MASK bit	function
0	0	clears the bit
0	1	sets the bit
1	0	leaves the bit unchanged
1	1	toggles (inverts) the bit

Remember that the AND is always applied before the XOR. Use this form with care: it is very easy to clear bits inadvertently with a badly chosen AND-MASK.

SET-MASK, CLR-MASK, and AT-ADDRESS can be specified in any order. The following are equivalent:

```
DEFINE <name>  SET-MASK <n>   CLR-MASK <m>  AT-ADDRESS <a> FOR-OUTPUT
DEFINE <name>  AT-ADDRESS <a> SET-MASK <n>  CLR-MASK <m>   FOR-OUTPUT
DEFINE <name>  CLR-MASK <m>   SET-MASK <n>  AT-ADDRESS <a> FOR-OUTPUT
```

Likewise, AND-MASK, XOR-MASK, and AT-ADDRESS can be specified in any order. But you *cannot* mix SET/CLR masks with AND/XOR masks.

Output trinaries can be used in state transitions and in procedural (Forth) code. **You are not required to use trinaries for the compound action in a state transition.** They are merely provided as a convenience.

An output trinary can be moved to Flash ROM with EEWORD:

```
DEFINE <name>  SET-MASK <n>   CLR-MASK <m>  AT-ADDRESS <a> FOR-OUTPUT
EEWORD
```

1.5.6. Defining Procedural Actions

For either test conditions or output actions, you may wish to specify procedural code. There is a form of the trinary declaraction that allows this:

---

[6] Currently, on the DSP5680x family, these operations are performed by reading memory, applying the logical operations, and then writing the result back to memory. But there is no guarantee that future versions of IsoMax, or versions for other processors, will be implemented in precisely the same way.

```
DEFINE <name>  PROC  ...procedural code...  END-PROC
```

When used to specify a test condition, the procedural (Forth) code should leave a true/false value on the stack. When used to specify an output action, the code should expect nothing from the stack, and when finished, leave nothing on the stack.

PROCs can be used within state transitions and in procedural (Forth) code. You are not required to use PROCs; they are provided as a convenience.[7]

These also can be moved to Flash ROM with EEWORD:

```
DEFINE <name>  PROC  ...procedural code...  END-PROC EEWORD
```

[7] `DEFINE ... PROC ... END-PROC` simply creates a normal Forth high-level ("colon") definition.

1.6. IsoMax Performance Monitoring

The IsoMax system is designed to execute user-defined state machines at a regular interval. This interval can be adjusted by the user with the `PERIOD` command. But how quickly can the state machine be executed? IsoMax provides tools to measure this, and also to handle the occasions when the state machine takes "too long" to process.

1.6.1. An Example State Machine

For the purposes of illustration, we'll use a state machine that blinks the green LED:[8]

```
LOOPINDEX CYCLE-COUNTER
DECIMAL 100 CYCLE-COUNTER END
1 CYCLE-COUNTER START

MACHINE SLOW_GRN

ON-MACHINE SLOW_GRN
 APPEND-STATE SG_ON
 APPEND-STATE SG_OFF

IN-STATE SG_ON
 CONDITION CYCLE-COUNTER COUNT
 CAUSES GRNLED OFF
 THEN-STATE SG_OFF
 TO-HAPPEN

IN-STATE SG_OFF
 CONDITION CYCLE-COUNTER COUNT
 CAUSES GRNLED ON
 THEN-STATE SG_ON
 TO-HAPPEN

SG_ON SET-STATE
INSTALL SLOW_GRN
```

This machine will execute at the default rate of `DECIMAL 50000 PERIOD`, or 100 Hz (since the clock rate is 5 MHz).

1.6.2. IsoMax Processing Time

Every time IsoMax processes your state machines, it measures the total number of clock cycles required. This is available to you in three variables:

`TCFAVG` This is a moving average of the measured processing time.[9] It is reported as a number of 5 MHz clock cycles.

---

[8] This example uses `LOOPINDEX` and `INSTALL`, and therefore requires IsoMax v0.36 or later.

[9] To be precise, `TCFAVG` is computed as the arithmetic mean of the latest measurement and the previous average, i.e., Tavg[n+1] = (Tmeasured + Tavg[n]) / 2.

TCFMIN This is the minimum measured processing time (in 5 MHz cycles). Note that this is *not* automatically reset when you install new state machines. Therefore, after installing new state machines, store a large value in TCFMIN to remove the old (false) minimum.

TCFMAX This is the maximum measured processing time (in 5 MHz cycles). This is *not* automatically reset when you change state machines. Therefore, after changing state machines, store a zero in TCFMAX to remove the old (false) maximum.

To see this, enter the following commands while the SLOW_GRN state machine is running:

```
DECIMAL 50000 TCFMIN !
0 TCFMAX !
TCFAVG ?
TCFMIN ?
TCFMAX ?
```

You may see an AVG and MIN time of about 630 cycles, and a MAX time near 1175 cycles.[10] With a 5 MHz clock, this corresponds to a processing time of about 126 usec (average) and 235 usec (maximum). The average is near the minimum because most of the time, the state machine is performing no action. Only once every 100 iterations does the CYCLE-COUNTER expire and force a change of LED state.

TCFAVG, TCFMIN, and TCFMAX return results in the same units used by PERIOD (counts of a 5 MHz clock). This means you can use TCFMAX to determine the safe lower bound of PERIOD. In this case, you could set PERIOD as low as 1175 decimal, and IsoMax would always have time to process the state machine.

1.6.3. Exceeding the Allotted Time

What if, in this example, PERIOD had been set to 1000 decimal? Most of the time, the state machine would be processed in less time, but once per second the LED transition would require more time than was allotted.

IsoMax will handle this gracefully by "skipping" clock interrupts as long as the state machine is still processing. With PERIOD set to 1000, an interrupt occurs every 200 usec. When the LED transition occurs, one interrupt will be skipped, and so there will be 400 usec (2000 cycles) between iterations of the state machine.

If this happens only rarely, it may not be of concern. But if it happens frequently, you may have a problem with your state machine, or you may have set PERIOD too low. To let you know when this is happening, IsoMax maintains an "overflow" counter:

---

[10] These times were measured on an IsoPod running the v0.37 kernel. With no state machines INSTALLed, the same kernel shows a TCFAVG of 88 cycles (17.6 usec). This represents the overhead to respond to a timer interrupt, service it, and perform an empty INSTALL list.

`TCFOVFLO` A variable, reset to zero when IsoMax is started, and incremented every time a clock interrupt occurs before IsoMax has completed state processing. (In other words, this tells you the number of "skipped" clock interrupts.)

You can see this in action by typing the following commands while the SLOW_GRN state machine is still running:

```
TCFOVFLO ?
DECIMAL 1000 PERIOD
TCFOVFLO ?
TCFOVFLO ?
TCFOVFLO ?
50000 PERIOD
TCFOVFLO ?
TCFOVFLO ?
```

Be sure to type these commands, and don't just upload them -- you need some time to elapse between commands so that you can see the overflow counter increase. After you change `PERIOD` back to 50000, the overflow counter will stop increasing.

1.6.4. Automatic Overflow Processing

If IsoMax overflows happen too frequently, you may wish your application to take some corrective action. You could write a program to monitor the value of `TCFOVFLO`. But IsoMax does this for you, and allows you to set an "alarm" value and an action to be performed:

`TCFALARM` A variable, set to zero when IsoMax is started. If set to a nonzero value, IsoMax will declare an "alarm" condition when the number of timer overflows (`TCFOVFLO`) reaches this value. If set to zero, timer overflows will be counted but otherwise ignored.

`TCFALARMVECTOR` A variable, set to zero when IsoMax is started. If set to a nonzero value, IsoMax will assume that this is the CFA of a Forth word to be executed when an "alarm" condition is declared. This Forth word should be stack-neutral, that is, it should consume no values from the stack, and should leave no values on the stack.

If set to zero, timer overflows will be counted but otherwise ignored.

Note that *both* of these values must be nonzero in order for alarm processing to take place. Be particularly careful that `TCFALARMVECTOR` is set to a valid address; if it is set to an invalid address it is likely to halt the IsoPod.

To continue with the previous example:

```
REDLED OFF
: TOO-FAST   REDLED ON  50000 PERIOD ;
' TOO-FAST CFA  TCFALARMVECTOR !
100 TCFALARM !
```

```
0 TCFOVFLO !
```

This defines a word TOO-FAST which is to be performed if too many overflows occur. TOO-FAST will turn on the red LED, and will also change the IsoMax period to a large (and presumably safe) value. The phrase ' TOO-FAST CFA returns the Forth CFA of the TOO-FAST word; this can be stored as the TCFALARMVECTOR. Finally, the alarm threshold is set to 100 overflows, and the overflow counter is reset.[11]

Now watch the LEDs after you type the command

```
1000 PERIOD
```

The slow blinking of the green LED will change to a rapid flicker for a few seconds. Then the red LED will come on and the green LED will return to a slow blink. This was caused by TOO-FAST being executed automatically when TCFOVFLO reached 100.

1.6.5. Counting IsoMax Iterations

It may be necessary for you to know how many times IsoMax has processed the state machine. IsoMax provides another variable to help you determine this:

TCFTICKS A variable, set to zero when IsoMax is started, and incremented on every IsoMax clock interrupt.

The frequency of the IsoMax clock interrupt is set by PERIOD; the default value is 100 Hz (50000 cycles of a 5 MHz clock). With this knowledge, you can use TCFTICKS for time measurement. With DECIMAL 50000 PERIOD, the variable TCFTICKS will be incremented 100 times per second.

Note that TCFTICKS is incremented *whether or not* an IsoMax overflow occurs. That is, it counts the number of IsoMax clock interrupts, *not* the number of times the state machine was processed. To compute the actual number of executions of the state machine, you must subtract the number of "skipped" clock interrupts, thus:

```
TCFTICKS @ TCFOVFLO @ -
```

[11] The test is for equality (TCFOVFLO=TCFALARM), not "greater than," to ensure that the alarm condition only happens once. The previous exercise left a large value in TCFOVFLO; if this is not reset to zero, the alarm won't occur until TCFOVFLO reaches 65535, "wraps around" back to zero, and then counts to 100.

1.7. Loop Indexes

A LOOPINDEX is an object that counts from a start value to an end value. Its name comes from the fact that it resembles the I index of a DO loop. However, LOOPINDEXes can be used anywhere, not just in DO loops. In particular, they can be used in IsoMax state machines to perform a counting function.

1.7.1. Defining a Loop Index

You define a LOOPINDEX just like you define a variable:

```
LOOPINDEX name
```

...where you choose the "name." For example,

```
LOOPINDEX CYCLE-COUNTER
```

Once you have defined a LOOPINDEX, you can specify a starting value, an ending value, and an optional step (increment) for the counter. For example, to specify that the counter is to go from 0 to 100 in steps of 2, you would type:

```
0 CYCLE-COUNTER START
100 CYCLE-COUNTER END
2 CYCLE-COUNTER STEP
```

You can specify these in any order. If you don't explicitly specify `START`, `END`, or `STEP`, the default values will be used. The default for a new counter is to count from 0 to 1 with a step of 1. So, if you want to define a counter that goes from 0 to 200 with a step of 1, all you have to change is the `END` value:

```
LOOPINDEX BLINK-COUNTER
200 BLINK-COUNTER END
```

If you use a negative `STEP`, the counter will count backwards. In this case the `END` value must be less than the `START` value!

You can change the `START`, `END`, and `STEP` values at any time, even when the counter is running.

1.7.2. Counting

The loopindex is incremented when you use the statement

```
name COUNT
```

For example,

```
CYCLE-COUNTER COUNT
```

COUNT will always return a truth value which indicates if the loopindex has *passed* its limit. If it has not, COUNT will return false (zero). If it has, COUNT will return true (nonzero), and it will also reset the loopindex value to the START value.

This truth value allows you to take some action when the limit is reached. This can be used in an IF..THEN statement:

```
CYCLE-COUNTER COUNT IF  GRNLED OFF  THEN
```

It can also be used as an IsoMax condition:

```
CONDITION CYCLE-COUNTER COUNT CAUSES GRNLED OFF ...
```

In this latter example, the loopindex will be incremented every time this condition is tested, but the CAUSES clause will be performed only when the loopindex reaches its limit.

Note that the limit test depends on whether STEP is positive or negative. If positive, the loopindex "passes" its limit when the count value + STEP value is *greater than* the END value. If negative, the loopindex passes its limit when the count value + STEP value is *less than* the END value.

In both cases, signed integer comparisons are used. **Be careful** that your loopindex limits don't result in an infinite loop! If you specify an END value of HEX 7FFF, and a STEP of 1, the loopindex will *never* exceed its limit, because in two's complement arithmetic, adding 1 to 7FFF gives -8000 hex -- a negative number, which is clearly *less* than 7FFF.

Also, be careful that you always use or discard the truth value left by COUNT. If you just want to increment the loopindex, without checking if it has passed its limit, you should use the phrase

```
CYCLE-COUNTER COUNT DROP
```

1.7.3. Using the Loopindex Value

Sometimes you need to know the value of the index while it is counting. This can be obtained with the statement

```
name VALUE
```

For example,

```
CYCLE-COUNTER VALUE
```

Sometimes you need to manually reset the count to its starting value, before it reaches the end of count. The statement

```
name RESET
```

will reset the index to its START value. For example,

```
CYCLE-COUNTER RESET
```

Remember that you *don't* need to explicitly RESET the loopindex when it reaches the end of count. This is done for you automatically. The loopindex "wraps around" to the START value, when the END value is passed.

1.7.4. A "DO loop"Example

This illustrates how a loopindex can be used to replace a DO loop in a program. This also illustrates the use of VALUE to get the current value of the loopindex.

```
LOOPINDEX BLINK-COUNTER
DECIMAL 20 BLINK-COUNTER END
2 BLINK-COUNTER STEP
: TEST   BEGIN  BLINK-COUNTER VALUE .  BLINK-COUNTER COUNT UNTIL ;
```

If you now type TEST, you will see the even numbers from 0 (the default START value) to 20 (the END value).[12] This is useful to show how the loopindex behaves with negative steps:

```
-2 BLINK-COUNTER STEP
40 BLINK-COUNTER START
BLINK-COUNTER RESET
TEST
```

This counts backwards by twos from 40 to 20. Note that, because we changed the START value of BLINK-COUNTER, we had to manually RESET it. Otherwise TEST would have started with the index value left by the previous TEST (zero), and it would have immediately terminated the loop (because it's less than the END value of 20).

1.7.5. An IsoMax Example

This example shows how a loopindex can be used within an IsoMax state machine, and also illustrates one technique to "slow down" the state transitions. Here we wish to blink the green LED at a rate 1/100 of the normal state processing speed. (Recall that IsoMax normally operates at 100 Hz; if we were to blink the LED at this rate, it would not be visible!)

```
LOOPINDEX CYCLE-COUNTER
DECIMAL 100 CYCLE-COUNTER END
1 CYCLE-COUNTER START

MACHINE SLOW_GRN

ON-MACHINE SLOW_GRN
```

[12] Forth programmers should note that the LOOPINDEX continues *up to and including* the END value, whereas a comparable DO loop continues only *up to* (but not including) its limit value.

```
 APPEND-STATE SG_ON
 APPEND-STATE SG_OFF

IN-STATE SG_ON
 CONDITION CYCLE-COUNTER COUNT
 CAUSES GRNLED OFF
 THEN-STATE SG_OFF
 TO-HAPPEN

IN-STATE SG_OFF
 CONDITION CYCLE-COUNTER COUNT
 CAUSES GRNLED ON
 THEN-STATE SG_ON
 TO-HAPPEN

SG_ON SET-STATE
INSTALL SLOW_GRN
```

Here the loopindex `CYCLE-COUNTER` counts from 1 to 100 in steps of 1. It counts in *either* state, and only when the count reaches its limit do we change to the other state (and change the LED). That is, the end-of-count `CAUSES` the LED action and the change of state. Since the counter is automatically reset after the end-of-count, we don't need to explicitly reset it in the IsoMax code.

1.7.6. Summary of Loopindex Operations

`LOOPINDEX name` Defines a "loop index" variable with the given `name`. For example,

```
LOOPINDEX COUNTER1
```

`START`
`END`
`STEP`
These words set the start value, the end value, or the step value (increment) for the given loop index. All of these expect an integer argument and the name of a loopindex variable. Examples:

```
  1 COUNTER1 START
100 COUNTER1 END
  3 COUNTER1 STEP
```

These can be specified in any order. If any of them is not specified, the default values will be used (START=0, END=1, STEP=1).

`COUNT` This causes the given loop index to increment by the STEP value, and returns a true or false value: true (-1) if the end of count was reached, false (0) otherwise. For example:

```
COUNTER1 COUNT
```

End of count is determined after the loop index is incremented, as follows: If STEP is positive, "end of count" is when the index is greater than the END value. If STEP is negative, "end of count" is when the index is less than the END value. Signed integer comparisons are used. In either case, when the end of count is reached, the loop index is reset to its START value.

`RESET` This word manually resets the given loop index to its START value. Example:

```
COUNTER1 RESET
```

`VALUE` This returns the current index value (counter value) of the given loop index. It will return a signed integer in the range -32768..+32767. For example:

```
COUNTER1 VALUE .        ...prints the loop index COUNTER1
```

1.8. Random Number Generator

(IsoMax version 0.75 and greater)

IsoMax includes a pseudo-random number generator that can be used to produce integer (single or double precision) and floating-point random numbers.

`RAND` returns a random single precision unsigned integer in the range 0 to 65535.

`DRAND` returns a random double precision unsigned integer in the range 0 to 2^{32}-1.

`FRAND` returns a random floating point value in the range 0.0 to 1.0.

`seed` is a double-precision integer variable (a `2VARIABLE`) which holds the "seed" of the random number generator. If you initialize this to a known value, you can generate a repeatable pseudo-random sequence. If you wish to generate a different sequence each time you use the IsoPod, you should initialize this to some random starting value.

Remember that this is a *pseudo*-random number generator. It has a reasonably long cycle (2^{32}) and is adequate for many applications (like rolling "electronic dice"). But for "serious" statistical modeling or simulation, you should write a more sophisticated random number generator.

When making "smaller" random numbers from `RAND` or `DRAND`, it is important to remember that the *most* significant bits are the most random. In other words, you should *not* mask off the high bits and use the low bits. Instead, you should use division to reduce the random number to a smaller range. For example, to simulate one throw of a die:

```
DECIMAL
: RAND6 ( -- n )  RAND 0  10922 UM/MOD  SWAP DROP ;
: TOSS  ( -- n )  BEGIN  RAND6  DUP 5 > WHILE DROP REPEAT ;
```

The random number (0 to 65535) is divided by 10922, using unsigned arithmetic, to produce a value in the range 0 to 5. The probabilities of getting 0 to 5 will be equal. Since there are six `RAND` values (65532-65535) which will produce a result of 6, `TOSS` is programmed to reject those, and will throw the die again if they occur.

1.8.1. Implementation Details

The pseudo-random number generator uses a simple "linear congruential" algorithm, as described by D. E. Knuth in *The Art of Computer Programming*. The recurrence relation for this generator is

$$X_{n+1} = (aX_n + c) \bmod m$$

where a=1664525, c=1, and m=2^{32} (the double-precision integer word size). These coefficients meet Knuth's theoretical requirements and have been found to produce well-distributed random numbers.[13]

1.9. Autostarting an IsoMax Application

1.9.1. The Autostart Search

When the IsoPod is reset, it searches the Program Flash ROM for an **autostart pattern**. This is a special pattern in memory which identifies an autostart routine. It consists of the value $A55A, followed by the address of the routine to be executed.

xx00: $A55A
xx01: address of routine

It must reside on an address within Program ROM which is a multiple of $400, i.e., $0400, $0800, $0C00, ... $7400, $7800, $7C00.

The search proceeds from $0400 to $7C00, and terminates when the *first* autostart pattern is found. This routine is then executed. If the routine exits, the IsoMax interpreter will then be started.

1.9.2. Writing an Application to be Autostarted

Any defined word can be installed as an autostart routine. For embedded applications, this routine will probably be an endless loop that never returns.

Here's a simple routine that reads characters from terminal input, and outputs their hex equivalent:

```
: MAIN   HEX BEGIN KEY . AGAIN ;   EEWORD
```

Note the use of `EEWORD` to put this routine into Flash ROM. An autostart routine must reside in Flash ROM, because when the IsoPod is powered off, the contents of RAM will be lost. If you install a routine in Program RAM as the autostart routine, the IsoPod will crash when you power it on. (To recover from such a crash, see "Bypassing the Autostart" below.)

Because this definition of `MAIN` uses a `BEGIN...AGAIN` loop, it will run forever. You can define this word from the keyboard and then type `MAIN` to try it out (but you'll have to reset the IsoPod to get back to the command interpreter). This is how you would write an application that is to run forever when the IsoPod is reset.

---

[13] Per Knuth, these coefficients were proposed by Lavaux and Janssens for 32-bit machines, and the corresponding generator measures well on the "spectral test" for random number distribution. See D. E. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*, Second Edition, Chapter 3, pp. 102-103 and 170-171.

You can also write an autostart routine that exits after performing some action. One common example is a routine that starts some IsoMax state machines. For this discussion, we'll use a version of MAIN that returns when an escape character is input:

```
HEX
: MAIN2   HEX BEGIN KEY DUP .  1B = UNTIL ;   EEWORD
```

In this example the loop will run continuously until the ESC character is received, then it exits normally. If this is installed as the autostart routine, when it exits, the IsoPod will proceed to start the IsoMax command interpreter.

1.9.3. Installing an Autostart Application

One the autostart routine is written, it can be installed into Flash ROM with the command

address AUTOSTART routine-name

This will build the autostart pattern in ROM. The *address* is the location in Flash ROM to use for the pattern, and must be a multiple of $400. Often the address $7C00 is used. This leaves the largest amount of Flash ROM for the application program, and leaves the option of later programming a new autostart pattern at a lower address. (Remember, the autostart search starts low and works up until the *first* pattern found, so an autostart at $7800 will override an autostart at $7C00.) So, for example, you could use

```
HEX 7C00 AUTOSTART MAIN2
```

to cause the word MAIN2 to be autostarted. (Note the use of the word HEX to input a hex number.)

Try this now, and then reset the IsoPod. You'll see that no "IsoMax" prompt is displayed. If you start typing characters at the terminal, you'll see the hex equivalents displayed. This will continue forever until you hit the ESC key, at which point the "IsoMax" prompt is displayed and the IsoPod will accept commands.

> **Note: starting with IsoMax version 0.61,** you do not need to provide an address for AUTOSTART. It will always use a default address for the autostart pattern. This example will still work, but you'll find the value 7C00 left on the stack because it wasn't used.

1.9.4. Saving the RAM data for Autostart

Power the IsoPod off, and back on, and observe that the autostart routine still works. Then press the ESC key to exit to the IsoMax command interpreter. Now try typing MAIN2. IsoMax doesn't recognize the word, even though you programmed it into Flash ROM! If you type WORDS you won't see MAIN2 in the listing. Why?

The reason is that some information about the words you have defined is kept in RAM[14]. If you just reset the board from MaxTerm, the RAM contents will be preserved. But if you power the board off and back on, the RAM contents will be lost, and IsoMax will reset RAM to known defaults. If you type `WORDS` after a power cycle, all you will see are the standard IsoMax words: all of your user-defined words are lost.

To prevent this from happening, you must save the RAM data to be restored on reset. This is done with the word `SAVE-RAM`:

```
SAVE-RAM
```

This can be done either just before, or just after, you use `AUTOSTART`. `SAVE-RAM` takes a "snapshot" of the RAM contents, and stores it in Data Flash ROM. Then, the next time you power-cycle the board, those preserved contents will be reloaded into RAM. This includes *both* the IsoMax system variables, and any variables or data structures you have defined.

Note: a simple reset will not reload the RAM. When the IsoPod is reset, it first checks to see if it has lost its RAM data. Only if the RAM has been corrupted -- as it is by a power loss -- will the IsoPod attempt to load the `SAVE-RAM` snapshot. (And only if there is no `SAVE-RAM` snapshot will it restore the factory defaults.) If you use MaxTerm to reset the IsoPod, the RAM contents will be preserved.

1.9.5. Removing an Autostart Application

Don't try to reprogram `MAIN2` just yet. Even though the RAM has been reset to factory defaults, `MAIN2` is still programmed into Flash ROM, and IsoMax doesn't know about it. In fact, if you try to redefine `MAIN2` at this point, you might crash the IsoPod, as it attempts to re-use Flash ROM which hasn't been erased. (To recover from this, see "Bypassing the Autostart," below.)

To completely remove all traces of your previous work, use the word `SCRUB`:

```
SCRUB
```

This will erase all of your definitions from Program Flash ROM -- including any `AUTOSTART` patterns which have been stored -- and will also erase any `SAVE-RAM` snapshot from Data Flash ROM. Basically, the word `SCRUB` restores the IsoPod to its factory-fresh state.

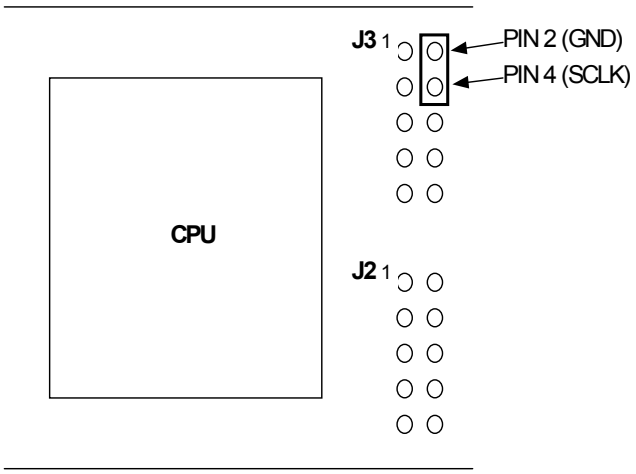
1.9.6. Bypassing the Autostart

What if your autostart routine locks up? If you can't get access to the IsoMax command interpreter, how do you `SCRUB` the application and restore the IsoPod to usability?

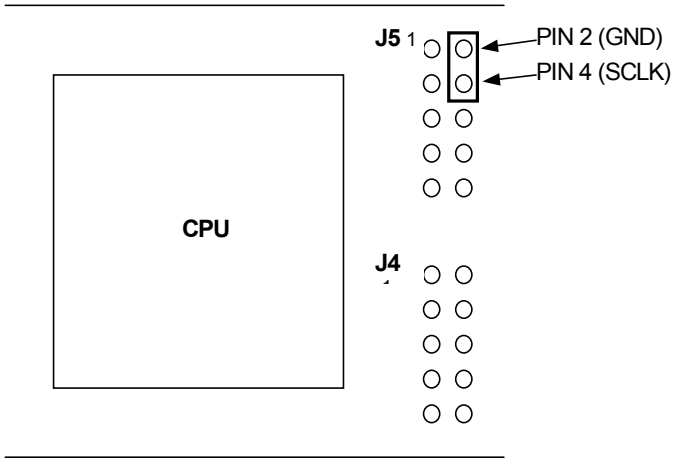
---

[14] To be specific, what is lost is the LATEST pointer, which always points to the last-defined word in the dictionary linked list. The power-up default for this is the last-defined word in the IsoMax kernel.

You can bypass the autostart search, and go directly to the IsoMax interpreter, by jumpering together pins 2 and 4 on connector J3, and then resetting the IsoPod. You can do this with a common jumper block:



IsoPod V1



IsoPod V2

This connects the SCLK/PE4 pin to ground. When the IsoPod detects this condition on reset, it does not perform the autostart search.

Note that this does *not* erase your autostart application or your `SAVE-RAM` snapshot from Flash ROM. These are still available for your inspection[15]. If you remove the jumper

[15] The IsoPod RAM will be reset to factory defaults instead of to the saved values, but you can still examine the `SAVE-RAM` snapshot in Flash ROM.

block and reset the IsoPod, it will again try to run your autostart application. (This can be a useful field diagnostic tool.)

To remove your application and start over, you'll need to use the `SCRUB` command. The steps are as follows:

1. Connect a terminal (or MaxTerm) to the RS-232 port.

2. Jumper pins 2 and 4 on J3.

3. Reset the IsoPod. You will see the "IsoMax" prompt.

4. Type the command `SCRUB` .

5. You can now remove the jumper from J3.

1.9.7. Summary

Use `EEWORD` to ensure that all of your application routines are in Flash ROM.

When your application is completely loaded, use `SAVE-RAM` to preserve your RAM data in Flash ROM.

Use `address AUTOSTART routine-name` to install your routine for autostarting. "address" must be a multiple of $0400 in empty Flash ROM; `HEX 7C00` is commonly used.

To clear your application and remove the autostart, use `SCRUB`. This restores the IsoPod to its factory-new state.

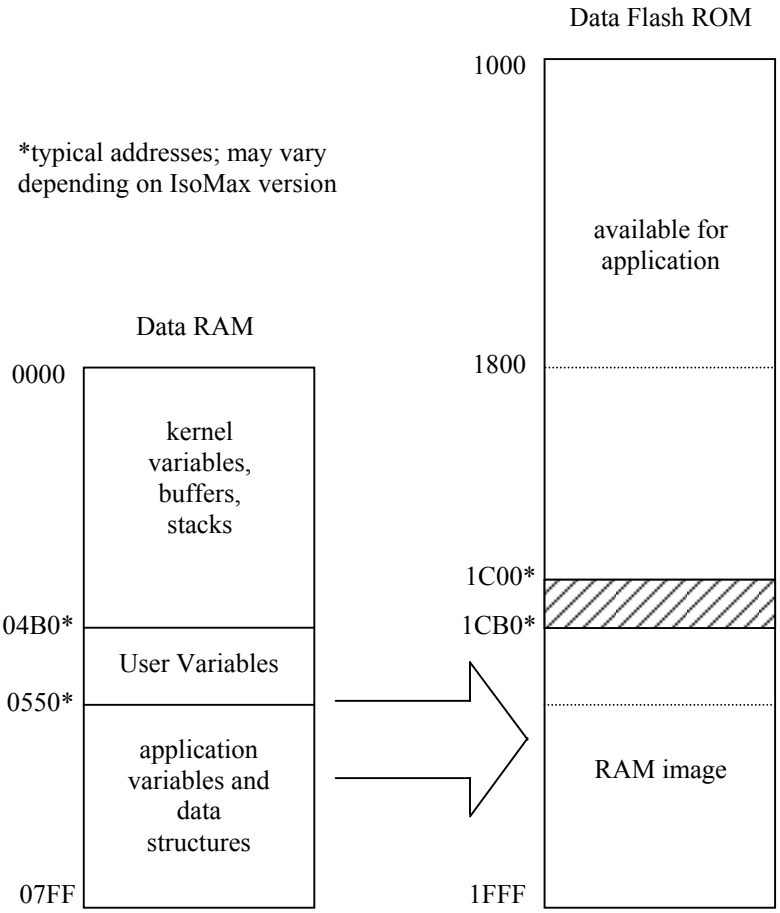
If the autostart application locks up, jumper together pins 2 and 4 of J3, and reset the IsoPod. This will give you access to the IsoMax command interpreter.

1.10. SAVE-RAM

The IsoPod contains 4K words of nonvolatile "Flash" data storage. This can be used to save system variables and your application variables so that they are automatically initialized when the IsoPod is powered up. This is done with the word `SAVE-RAM`.

1.10.1. Data Memory Map

The internal RAM of the IsoPod is divided into three regions: kernel buffers, User Variables, and application variables.



Kernel buffers include the stacks, working "registers," and other scratch data that are used by the IsoMax interpreter. These are considered "volatile" and are always cleared when the IsoPod is powered up. These are also private to IsoMax and not available to you.

"User Variables" are IsoMax working variables which you may need to examine or change. These include such values as the current number base (`BASE`), the current ROM

and RAM allocation pointers, and the Terminal Input Buffer. This region also includes RAM for the IsoMax state machine and the predefined IsoPod I/O objects.

Application data is whatever variables, objects, and buffers you define in your application program. This can extend up to the end of RAM (address 07FF hex in the IsoPod).

1.10.2. Saving the RAM image

The word `SAVE-RAM` copies the User Variables and application data to the *end* of Data Flash ROM. All of internal RAM, starting at the first User Variable (currently `C/L`) and continuing to the end of RAM, is copied to corresponding addresses in the Flash ROM.

Note that this will copy all `VARIABLE`s and the RAM contents of all objects, but it will *not* copy the stacks.

Normally you will use `SAVE-RAM` to take a "snapshot"of your RAM data when all your variables are initialized and your application is ready to run.

1.10.2.1. Flash erasure

Because the `SAVE-RAM` uses Flash memory, it must erase the Flash ROM before it can copy to it. This is automatically done by `SAVE-RAM`, and you need not perform any explicit erase function. However, you should be aware that `SAVE-RAM` will erase more Flash ROM than is needed for the RAM image.

Flash ROM is erased in "pages" of 256 words each. To ensure that all of the RAM image is erased, `SAVE-RAM` must erase starting at the next lower page boundary. A page boundary address is always of the form $XX00 (the low eight bits are zero). So, in the illustrated example, Flash ROM is erased starting at address $1C00.

If you use Data Flash ROM directly in your application, you can be sure that your data will be safe if you restrict your usage to addresses $1000-$17FF. Some of the space above $1800 is currently unused, but this is not guaranteed for future IsoMax releases.

1.10.3. Restoring the RAM image

The IsoPod will automatically copy the saved RAM image from Flash ROM back to RAM when it is first powered up. This will occur before your application program is started. So, you can use `SAVE-RAM` to create an "initial RAM state" for your application.

If the IsoPod is reset and the RAM contents appear to be valid, the saved RAM image will *not* be used. This may happen if the IsoPod receives a hardware reset signal while power is maintained. Usually this is the desired behavior.

1.10.3.1. Restoring the RAM image manually

You can force RAM to be copied from the saved image by using `RESTORE-RAM`. This does exactly the reverse of `SAVE-RAM`: it copies the contents of Data Flash ROM to Data RAM. The address range copied is the same as used by `SAVE-RAM`.

So, if your application needs RAM to be initialized on every hardware reset (and not just on a power failure), you can put `RESTORE-RAM` at the beginning of your autostart routine.

Note: do not use `RESTORE-RAM` *if* `SAVE-RAM` *has not been performed.* This will cause invalid data to be written to the User Variables (and to your application variables as well), which will almost certainly crash the IsoPod. For most applications it is sufficient, and safer, to use the default RAM restore which is built into the IsoPod kernel.

1.11. IsoPod™ Reset Sequence

The IsoPod employs a flexible initialization that gives you many options for starting and running application programs. Sophisticated applications can elect to run with or without IsoMax, and with the default or custom processor initialization. This requires some knowledge of the steps that the IsoPod takes upon a processor reset:

1. Perform basic CPU initialization. This includes the PLL clock generator and the RS232 serial port.

2. Do the `QUICK-START` routine. If a `QUICK-START` vector is present in RAM, execute the corresponding routine. `QUICK-START` is designed to be used before any other startup code, normally just to provide some additional initialization. In particular, this is performed before RAM is re-initialized. This gives you the opportunity to save any RAM status, for example on the occurrence of a watchdog reset. Note that a power failure which clears the RAM will also clear the `QUICK-START` vector.

3. Stop IsoMax. This is in case of a "software reset" that would otherwise leave the timer running.

4. Check for "autostart bypass." Configure the SCLK/PE4 pin as an input with pullup resistor. If the SCLK/PE4 pin then reads a continuous "0" (ground level) for 1 millisecond, skip the autostart sequence and "coldstart" the IsoPod. This will initialize RAM to factory defaults and start the IsoMax interpreter.

> This is intended to recover from a situation where an autostart application locks up the IsoPod. Simply jumper the SCLK/PE4 pin to ground, and reset the IsoPod. This will reset the RAM and start the interpreter, but please note that it will *not* erase any Flash ROM. Flash ROM can be erased with the `SCRUB` command from the IsoMax interpreter.
>
> This behavior should be kept in mind when designing hardware around the IsoPod. If the IsoPod is installed as an SPI master, or if the SCLK/PE4 pin is used as a programmed output, there will be no problem. If the IsoPod is installed as an SPI slave, the presence of SPI clock pulses will not cause a coldstart, but a coldstart *will* happen if SCLK is held low in the "idle" state and a CPU reset occurs. For this reason, if the IsoPod is an SPI slave, we recommend configuring the SPI devices with CPOL=1, so the "idle" state of SCLK is high. If the SCLK/PE4 pin is used as a programmed input, avoid applications where this pin might be held low when a CPU reset occurs.

If SCLK/PE4 is *not* grounded, proceed with the autostart sequence.

5. Check the contents of RAM and initialize as required.

a. If the RAM contents are valid[16], use them. This will normally be the case if the CPU is reset with no power cycle, e.g., reset by MaxTerm, a watchdog, or an external reset signal.

b. If the RAM contents are invalid, load the `SAVE-RAM` image from Data Flash ROM. If this RAM image is valid, use it. This gives you a convenient method to initialize your application RAM.

c. If the Flash ROM contents are invalid, then reinitialize RAM to factory defaults. Note that this will reset the dictionary pointer but will *not* erase any Flash ROM.

6. Look for a "boot first" routine. Search for an $A44A pattern in Program Flash ROM. The search looks at 1K ($400) boundaries, starting at Program address $400 and proceeding to $7C00. If found, execute the corresponding "boot first" routine. IsoMax is *not* running at this point.

a. If the "boot first" routine never exits, only it will be run.

b. If the "boot first" routine exits, or if no $A44A pattern is found, continue the autostart sequence.

7. Start IsoMax with an "empty" list of state machines. After this, you can begin `INSTALL`ing state machines. Any state machines `INSTALL`ed before this point will be disabled.

8. Look for an "autostart" routine. Search for an $A55A pattern in Program Flash ROM. The search looks at 1K ($400) boundaries, starting at Program address $400 and proceeding to $7C00. If found, execute the corresponding "autostart" routine.

a. If the "autostart" routine never exits, only it will be run. (Of course, any IsoMax state machines `INSTALL`ed by this routine will also run.)

b. If the "autostart" routine exits, or if no $A55A pattern is found, start the IsoMax interpreter.

1.11.1. In summary:

Use the `QUICK-START` vector if you need to examine uninitialized RAM, or for chip initialization which must occur immediately.

Use an $A44A "boot first" vector for initialization which must *precede* IsoMax activation, but which needs initialized RAM.

Use an $A55A "autostart" vector to install IsoMax state machines, and for your main application program.

To bypass the autostart sequence, jumper SCLK/PE4 to ground.

---

[16] RAM is considered "valid" if the program dictionary pointer is within the Program Flash ROM address space, the version number stored in RAM matches the kernel version number, and the SYSTEM-INITIALIZED variable contains the value $1234.

1.12. 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.

1.12.1. 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
```

1.12.2. 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

1.12.3. 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.*

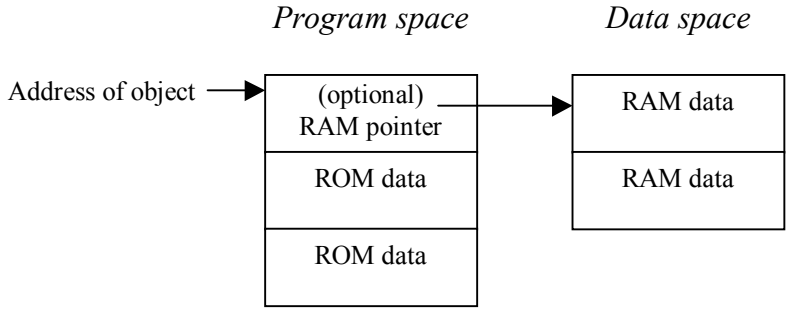
1.12.4. Referencing an object

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.

1.12.5. 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 although `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.

1.12.6. 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`.

1.13. Machine Code Programming

IsoMax allows individual words to be written in machine code as well as "high-level" language code. Such words are indistinguishable in function from high-level words, and may be used freely in application programs and state machines.

1.13.1. Assembler Programming

The IsoPod uses the Motorola DSP56F805 microprocessor. The machine language of this processor is described in Motorola's *DSP56800 16-Bit Digital Signal Processor Family Manual,* available at

<http://e-www.motorola.com/brdata/PDFDB/docs/DSP56800FM.pdf>.

IsoMax does *not* include a symbolic assembler for this processor. You must use an external assembler to convert your program to the equivalent hexadecimal machine code, and then insert these numeric opcodes and operands into your IsoMax source code.[17] For an example, let's use an assembler routine to stop Timer D2:

```
; Timer/Counter
; -------------
; Timer control register
;  000x xxxx xxxx xxxx = no count
andc    #$1FFF,X:$0D76  ; TMRD2_CTRL

; Timer status & control register
; Clear TCF flag, clear interrupt enable flag
bfclr   #$8000,X:$0D77  ; TMRD2_SCR  clear TCF
bfclr   #$4000,X:$0D77  ; TMRD2_SCR  clear TCFIE
```

Translated to machine code, this is:

```
80F4    andc    #$1FFF,X:$0D76
0D76
E000
80F4    bfclr   #$8000,X:$0D77
0D77
8000
80F4    bfclr   #$4000,X:$0D77
0D77
4000
```

[17] If you wish to translate your programs manually to machine code, a summary chart of DSP56800 instruction encoding is given at the end of this manual.

To compile this manually into an IsoMax word, you must append each hexadecimal value to the dictionary with the `P,` operator. (The "P" refers to Program space, where all machine code must reside.) You can put more than one value per line:

```
80F4 P, 0D76 P, E000 P,
80F4 P, 0D77 P, 8000 P,
80F4 P, 0D77 P, 4000 P,
```

All that remains is to add this as a word to the IsoMax dictionary, and to return from the assembler code to IsoMax. There are three ways to do this: with `CODE`, `CODE-SUB`, and `CODE-INT`.

1.13.2. CODE functions

The special word `CODE` defines a machine language word as follows:

```
CODE word-name

    (machine language for your word)

    (machine language for JMP NEXT)

END-CODE
```

Machine code words that are created with `CODE` must return to IsoMax by performing a jump to the special address `NEXT`. *In IsoMax versions 0.52 and higher, this is address $0080. Earlier versions of IsoMax do not support NEXT and you must use CODE-SUB, described below, to write machine code words.*

An absolute jump instruction is $E984. Thus a `JMP NEXT` translates to $E984 $0080, and our example `STOP-TIMERD2` word could be written as follows:

```
HEX
CODE STOP-TIMERD2
    80F4 P, 0D76 P, E000 P,
    80F4 P, 0D77 P, 8000 P,
    80F4 P, 0D77 P, 4000 P,
    E984 P, 0080 P, ( JMP NEXT )
END-CODE
```

Remember, this example will only work on recent versions of IsoMax (0.52 or later).

1.13.3. CODE-SUB functions

The special word `CODE-SUB` is just like `CODE`, except that the machine code returns to IsoMax with an ordinary `RTS` instruction. This can be useful if you need to write a machine code routine that can be called both from IsoMax and from other machine code

routines. It's also useful if the NEXT address is not available (as in IsoMax versions prior to 0.52). The syntax is similar to CODE:

```
CODE-SUB word-name

    (machine language for your word)

    (machine language for RTS)

END-CODE
```

An RTS instruction is $EDD8, so STOP-TIMERD2 could be written with CODE-SUB as follows:

```
HEX
CODE-SUB STOP-TIMERD2
    80F4 P, 0D76 P, E000 P,
    80F4 P, 0D77 P, 8000 P,
    80F4 P, 0D77 P, 4000 P,
    EDD8 P, ( RTS )
END-CODE
```

This example will work in all versions of IsoMax.

1.13.4. CODE-INT functions

CODE-INT is just like CODE-SUB, except that the machine code returns to IsoMax with an RTI (Return from Interrupt) instruction, $EDD9. This is useful if you need to write a machine code interrupt handler that can also be called directly from IsoMax. *CODE-INT is only available on IsoMax versions 0.52 and later.*

```
HEX
CODE-INT STOP-TIMERD2
    80F4 P, 0D76 P, E000 P,
    80F4 P, 0D77 P, 8000 P,
    80F4 P, 0D77 P, 4000 P,
    EDD9 P, ( RTI )
END-CODE
```

To obtain the address of the machine code after it is compiled, use the phrase

```
' word-name CFA 2+
```

Note: if you are using EEWORD to put this new word into Flash ROM, use EEWORD *before* trying to obtain the address of the machine code. EEWORD will change this address.

1.13.5. Register Usage

In the current version of IsoMax software, all DSP56800 address and data registers may be used in your CODE and CODE-SUB words. You need not preserve R0-R3, X0, Y0, Y1, A, B, or N. Do not change the "mode" registers M01 or OMR, and do not change the stack pointer SP.

Future versions of IsoMax may add more restrictions on register use. If you are concerned about compatibility with future kernels, you should save and restore all registers that your machine code will use.

CODE-INT words are expected to be called from interrupts, and so they should save any registers that they use.

1.13.6. Calling High-Level Words from Machine Code

You can call a high-level IsoMax word from within a machine-code subroutine. This is done by calling the special subroutine ATO4 with the address of the word you want to execute.[18] This address must be a Code Field Address (CFA) and is obtained with the phrase

```
' word-name CFA
```

This address must be passed in register R0. You can load a value into R0 with the machine instruction $87D0, $xxxx (where xxxx is the value to be loaded).

The address of the ATO4 routine can be obtained from a constant named ATO4. You can use this constant directly when building machine code. The opcode for a JSR instruction is $E9C8, $aaaa where aaaa is an absolute address. So, to write a CODE-SUB routine that calls the IsoMax word DUP, you could write:

```
HEX
CODE-SUB NEWDUP
    87D0 P, ' DUP CFA P, ( move DUP CFA to R0 )
    E9C8 P, ATO4 P,      ( JSR ATO4 )
    EDD8 P,          ( RTS )
END-CODE
```

Observe that the phrases ' DUP CFA and ATO4 are used *within* the CODE-SUB to generate the proper addresses where required.

[18]The name ATO4 comes from "Assembler to Forth" and refers to the Forth underpinnings of IsoMax.

1.14. Using CPU Interrupts in the IsoPod

This applies to IsoPod kernel v0.38 and later.

1.14.1. Interrupt Vectors in Flash ROM

The DSP56F805 processor used in the IsoPod supports 64 interrupt vectors, in the first 128 locations of Flash ROM. Each vector is a two-word machine instruction, normally a JMP instruction to the corresponding interrupt routine. When an interrupt occurs, the CPU jumps directly to the appropriate address ($00-$7E) in the vector table.

Since this vector table is part of the IsoPod kernel, it cannot be altered by the user. Also, some interrupts are required for the proper functioning of the IsoPod, and these vectors must never be changed. So the IsoPod includes a "user" vector table at the high end of Flash ROM (addresses $7D80-7DFE). This is exactly the same as the "kernel" vector table, except that certain "reserved for IsoPod" interrupts have been excluded. The user vector table can be programmed, erased, and reprogrammed freely by the user, as long as suitable precautions are taken.

1.14.2. Writing Interrupt Service Routines

Interrupt service routines must be written in DSP56F805 machine language, and must end with an RTI (Return from Interrupt) instruction. Some peripherals will have additional requirements; for example, many interrupt sources need to be explicitly cleared by the interrupt service routine. For more information about interrupt service routines, refer to the Motorola DSP56800 16-Bit Digital Signal Processor Family Manual (Chapter 7), and the Motorola DSP56F801/803/805/807 16-Bit Digital Signal Processor User's Manual.

You should be aware that the IsoPod uses certain channels in the Interrupt Priority controller:

> The IsoMax Timer (Timer C3[19]) is assigned to Interrupt Priority Channel 3.
> SCI (RS-232) and SPI serial I/O is assigned to Interrupt Priority Channel 4.
> The I/O Scheduling Timer[20] is assigned to Interrupt Priority Channel 5.

These channels may be shared by other peripherals. However, it is important to remember that these channels are *enabled* by the IsoMax kernel after a reset, and must never be disabled. You should not use the corresponding bits in the Interrupt Priority Register as interrupt enable/disable bits.

---

[19] Timer D3 on IsoPods before version 0.65.
[20] Version 0.69 and later.

Interrupt channels 0, 1, 2, and 6 are reserved for your use. The IsoMax kernel does not use them, and you may assign, enable, or disable them freely. Channel 0 has the lowest priority, and 6 the highest.[21]

1.14.3. The User Interrupt Vector Table

The user vector table is identical to the kernel (CPU) vector table, except that it starts at address $7D80 instead of address $0. Each interrupt vector is two words in this table, sufficient for a machine language jump instruction. For all interrupts which are not reserved by IsoMax, the kernel vector table simply jumps to the corresponding location in the user vector table. (Remember that this adds the overhead of one absolute jump instruction -- 6 machine clock cycles -- to the interrupt service.)

Note: IsoPod kernels version 0.37 and earlier do not support a user vector table.

Note: This table is subject to change. Future versions of the IsoPod software may reserve more of these interrupts for internal use, as more I/O functions are added to the IsoPod kernel.

Interrupt Number	User Vector Address	Kernel Vector Address	Description
0		$00	reset - *reserved for IsoPod*
1	$7D82	$02	COP Watchdog reset
2	$7D84	$04	reserved by Motorola
3		$06	illegal instruction - *reserved for IsoPod*
4	$7D88	$08	Software interrupt
5	$7D8A	$0A	hardware stack overflow
6	$7D8C	$0C	OnCE Trap
7	$7D8E	$0E	reserved by Motorola
8	$7D90	$10	external interrupt A
9	$7D92	$12	external interrupt B
10	$7D94	$14	reserved by Motorola
11	$7D96	$16	boot flash interface
12	$7D98	$18	program flash interface
13	$7D9A	$1A	data flash interface
14	$7D9C	$1C	MSCAN transmitter ready
15	$7D9E	$1E	MSCAN receiver full
16	$7DA0	$20	MSCAN error
17	$7DA2	$22	MSCAN wakeup
18	$7DA4	$24	reserved by Motorola
19		$26	GPIO E - *reserved for IsoPod*
20	$7DA8	$28	GPIO D
21	$7DAA	$2A	reserved by Motorola
22		$2C	GPIO B - *reserved for IsoPod*
23		$2E	GPIO A - *reserved for IsoPod*
24		$30	SPI transmitter empty - *reserved for IsoPod*

[21] Use channel 6 only for critically-urgent interrupts, since it will take priority over channels 4 and 5, both of which require prompt service.

Interrupt Number	User Vector Address	Kernel Vector Address	Description
25		$32	SPI receiver full/error - *reserved for IsoPod*
26	$7DB4	$34	Quad decoder #1 home
27	$7DB6	$36	Quad decoder #1 index pulse
28	$7DB8	$38	Quad decoder #0 home
29	$7DBA	$3A	Quad decoder #0 index pulse
30	$7DBC	$3C	Timer D Channel 0
31	$7DBE	$3E	Timer D Channel 1
32	$7DC0	$40	Timer D Channel 2
33	$7DC2	$42	Timer D Channel 3
34	$7DC4	$44	Timer C Channel 0
35	$7DC6	$46	Timer C Channel 1
36		$48	Timer C Channel 2 - *reserved for IsoPod*
37		$4A	Timer C Channel 3 - *reserved for IsoPod*
38	$7DCC	$4C	Timer B Channel 0
39	$7DCE	$4E	Timer B Channel 1
40	$7DD0	$50	Timer B Channel 2
41	$7DD2	$52	Timer B Channel 3
42	$7DD4	$54	Timer A Channel 0
43	$7DD6	$56	Timer A Channel 1
44	$7DD8	$58	Timer A Channel 2
45	$7DDA	$5A	Timer A Channel 3
46	$7DDC	$5C	SCI #1 Transmit complete
47		$5E	SCI #1 transmitter ready - *reserved for IsoPod*
48	$7DE0	$60	SCI #1 receiver error
49		$62	SCI #1 receiver full - *reserved for IsoPod*
50	$7DE4	$64	SCI #0 Transmit complete
51		$66	SCI #0 transmitter ready - *reserved for IsoPod*
52	$7DE8	$68	SCI #0 receiver error
53		$6A	SCI #0 receiver full - *reserved for IsoPod*
54	$7DEC	$6C	reserved by Motorola
55	$7DEE	$6E	ADC A Conversion complete
56	$7DF0	$70	reserved by Motorola
57	$7DF2	$72	ADC A zero crossing/error
58	$7DF4	$74	Reload PWM B
59	$7DF6	$76	Reload PWM A
60	$7DF8	$78	PWM B Fault
61	$7DFA	$7A	PWM A Fault
62	$7DFC	$7C	PLL loss of lock
63	$7DFE	$7E	low voltage detector

1.14.4. Clearing the User Vector Table

Since the user vector table is at the high end of Flash ROM, it will be erased by the SCRUB command (which erases all of the user-programmable Flash ROM).

If you wish to erase *only* the user vector table, you should use the command

```
HEX 7D00 PFERASE
```

This will erase 256 words of Program Flash ROM, starting at address 7D00. In other words, this will erase locations 7D00-7DFF, which includes the user vector table. Because of the limitations of Flash ROM, you cannot erase a smaller segment -- you must erase 256 words. However, this is at the high end of Flash ROM and is unlikely to affect your application program, which is built upward from low memory.

When Flash ROM is erased, all locations read as $FFFF. This is an illegal CPU instruction. So it is very important that you install an interrupt vector *before* you enable the corresponding interrupt! If you enable a peripheral interrupt when no vector has installed, you will cause an Illegal Instruction trap and the IsoPod will reset.[22]

1.14.5. Installing an Interrupt Vector

Once the Flash ROM has been erased, you can write data to it with the `PF!` operator. Each location can be written only once, and must be erased before being written with a different value.[23]

For example, this will program the low-voltage-detect interrupt to jump to address zero. (This will restart the IsoPod, since address zero is the reset address.)

```
HEX  E984 7DFE PF!  0 7DFF PF!
```

E984 is the machine language opcode for an absolute jump; this is written into the first word of the vector. The destination address, 0, is written into the second word. Because these addresses are in Flash ROM, you must use the `PF!` operator. An ordinary `!` operator will not work.

1.14.6. Precautions when using Interrupts

1. An unprogrammed interrupt vector will contain an FFFF instruction, which is an illegal instruction on the DSP56F805. Don't enable an interrupt until *after* you have installed its interrupt vector.

2. Remember that most interrupts must be cleared at the source before your service routine Returns from Interrupt (with an RTI instruction). If you forget to clear the interrupt, you may end in an infinite loop.

3. Remember that `SCRUB` will erase all vectors in the user table. Be sure to disable *all* of the interrupts that you have enabled, before you use `SCRUB`.

---

[22] This is why the "illegal instruction" interrupt is reserved for IsoMax. If it were vectored to the user table, and you did not install a vector for it, the attempt to service an illegal instruction would cause yet another illegal instruction, and the CPU would lock up.

[23] Strictly speaking, you can write a Flash ROM location more than once, but you can only change "1" bits to "0." Once a bit has been written as "0", you need to erase the ROM page to return it to a "1" state.

4. You cannot erase a single vector in the user table. You must use `HEX 7D00 PFERASE` to erase the entire table. As with `SCRUB`, be sure to disable all of your interrupt sources first.

5. Do *not* use the global interrupt enable (bits I1 and I0 in the Status Register) to disable your peripheral interrupts. This will also shut off the interrupts that are used by IsoMax, and the IsoPod will likely halt.

6. It *is* permissible to disable interrupts globally for extremely brief periods -- on the order of a few machine instructions -- in order to perform operations that mustn't be interrupted. But this may affect critical timing within IsoMax, and is generally discouraged.

7. You can perform the action of an IsoPod reset by jumping to absolute address zero. But note that, unlike a true hardware reset, this will *not* disable any interrupt sources that you may have enabled.

1.15. Interrupt Handlers in High-Level Code

Interrupt handlers must be written in machine code. However, you can write a machine code "wrapper" that will call a high-level IsoMax word to service an interrupt. This application note describes how. You may find it useful to refer to the previous sections *Machine Code Programming* and *Using CPU Interrupts in the IsoPod*.

1.15.1. How it Works

The machine code routine below works by saving all the registers used by IsoMax, and then calling the ATO4 routine to run a high-level IsoMax word. The high-level word returns to the machine code, which restores registers and returns from the interrupt.

```
HEX 0041 CONSTANT WP

CODE-SUB INT-SERVICE
DE0B P,           \  LEA  (SP)+
D00B P,           \  MOVE X0,X:(SP)+
D10B P,           \  MOVE Y0,X:(SP)+
D30B P,           \  MOVE Y1,X:(SP)+
D08B P,           \  MOVE A0,X:(SP)+
D60B P,           \  MOVE A1,X:(SP)+
D28B P,           \  MOVE A2,X:(SP)+
D18B P,           \  MOVE B0,X:(SP)+
D70B P,           \  MOVE B1,X:(SP)+
D38B P,           \  MOVE B2,X:(SP)+
D80B P,           \  MOVE R0,X:(SP)+
D90B P,           \  MOVE R1,X:(SP)+
DA0B P,           \  MOVE R2,X:(SP)+
DB0B P,           \  MOVE R3,X:(SP)+
DD0B P,           \  MOVE N,X:(SP)+
DE8B P,           \  MOVE LC,X:(SP)+
DF8B P,           \  MOVE LA,X:(SP)+
F854 P, OBJREF P, \  MOVE X:OBJREF,R0
FA54 P, WP P,     \  MOVE X:WP,R2
D80B P,           \  MOVE R0,X:(SP)+
DA1F P,           \  MOVE R2,X:(SP)    ; Note no increment on last push!
87D0 P, xxxx P,   \  MOVE #$XXXX,R0    ; This is the CFA of the word to execute
E9C8 P, ATO4 P,   \  JSR  ATO4         ; do that Forth word
FA1B P,           \  MOVE X:(SP)-,R2   ; restore the saved wp
F81B P,           \  MOVE X:(SP)-,R0   ; restore the saved objref
FF9B P,           \  MOVE X:(SP)-,LA
DA54 P, WP P,     \  MOVE R2,X:FWP
D854 P, OBJREF P, \  MOVE R0,X:OBJREF
FE9B P,           \  MOVE X:(SP)-,LC
FD1B P,           \  MOVE X:(SP)-,N
FB1B P,           \  MOVE X:(SP)-,R3
FA1B P,           \  MOVE X:(SP)-,R2
F91B P,           \  MOVE X:(SP)-,R1
F81B P,           \  MOVE X:(SP)-,R0
F39B P,           \  MOVE X:(SP)-,B2
F71B P,           \  MOVE X:(SP)-,B1
```

```
F19B P,           \  MOVE X:(SP)-,B0
F29B P,           \  MOVE X:(SP)-,A2
F61B P,           \  MOVE X:(SP)-,A1
F09B P,           \  MOVE X:(SP)-,A0
F31B P,           \  MOVE X:(SP)-,Y1
F11B P,           \  MOVE X:(SP)-,Y0
F01B P,           \  MOVE X:(SP)-,X0
EDD9 P,           \  RTI
END-CODE
```

The only registers that are saved automatically by the processor are PC and SR. *All* other registers that will be used must be saved manually. To allow a high-level routine to execute, we must save R0-R3, X0, Y0, Y1, A, B, N, LC, and LA. Two registers that need not be saved are M01 and OMR, because these registers are never used or changed by IsoMax. We must also save the two variables `WP` and `OBJREF`, which are used by the IsoMax interpreter and object processor.

Since the DSP56F805 processor does not have a "pre-increment" address mode, the first push must be *preceded* by a stack pointer increment, LEA (SP)+, and the last push must *not* increment SP.

The instruction ordering may seem peculiar; this is because a MOVE to an address reigster (Rn) has a one-instruction delay. So we always interleave another unrelated instruction after a MOVE x, Rn. Note also the use of the symbols `ATO4` and `OBJREF` to obtain addresses. The variable `WP` is located at hex address 0041 in current IsoMax kernels, and this is defined as a constant for readability.

The value shown as "xxxx" in the listing above is where you must put the Code Field Address (CFA) of the desired high-level word. You can obtain this address with the phrase

```
' word-name CFA
```

1.15.2. Use of Stacks

The interrupt routine will use the same Data and Return stacks as the IsoMax command interpreter, that is, the "main" program.[24] Normally this is not a problem, because pushing new data onto a stack does not affect the data which is already there. However, you must take care that your interrupt handler leaves the stacks as it found them – that is, does not leave any extra items on the stack, or consume any items that were already there. A stack imbalance in an interrupt handler is a very quick way to crash the IsoPod.

1.15.3. Use of Variables

Some high-level words use temporary variables and buffers which are not saved when an interrupt occus. One example is the numeric output functions (`. D. F.` and the like).

[24]The IsoMax state machine uses an independent set of stacks.

You should not use these words within your interrupt routine, since this will corrupt the variables that might be used by the main program.

1.15.4. Re-Entrancy

To avoid re-entrancy problems, it is best to *not* re-enable interrupts within your high-level interrupt routine. Interrupts will be re-enabled automatically by the `RTI` instruction, when your routine has finished its processing.

You must of course be sure to clear the interrupt source in your high-level service routine. If you fail to do so, when the `RTI` instruction is executed, a new interrupt will instantly occur, and your program will be stuck in an infinite loop of interrupts.

1.15.5. Example: Millisecond Timer

This example uses Timer D2 to increment a variable at a rate of once per millisecond. After loading the entire example, you can use `START-TMRD2` to initialize the timer, set up the interrupt controller for that timer, and enable the interrupt. From that point on, the variable `TICKS` will be incremented on every interrupt. You can fetch the `TICKS` variable in your main program (or from the command interpreter).

The high-level interrupt service routine is `INT-SERVICE`. It does only two things. First it clears the interrupt source, by clearing the TCF bit in the Timer D2 Status and Control Register. Then it increments the variable `TICKS`. As a rule, interrupt service routines should be as short and simple as possible. Remember, no other processing takes place while the interrupt is being serviced.

You can stop the timer interrupt with `STOP-TMRD2`.

```
\ Count for 1 msec at 5 MHz timer clock
DECIMAL 5000 CONSTANT TMRD2_COUNT EEWORD
HEX

0C00 CONSTANT IOBASE EEWORD   \ use 1000 for ServoPod

\ Timer D2 registers
IOBASE 0170 + CONSTANT TMRD2_CMP1 EEWORD
IOBASE 0173 + CONSTANT TMRD2_LOAD EEWORD
IOBASE 0176 + CONSTANT TMRD2_CTRL EEWORD
IOBASE 0177 + CONSTANT TMRD2_SCR  EEWORD

\ GPIO interrupt control register
FFFB CONSTANT GPIO_IPR   EEWORD
2000 CONSTANT GPIO_IPL_2 EEWORD   \ bit which enables Channel 2 IPL

\ Interrupt vector & control.
\ Timer D channel 2 is vector 36, IRQ table address $48
0040 7D80 + CONSTANT TMRD2_VECTOR  EEWORD

\ Timer D channel 2 is controlled by Group Priority Register GPR8, bits
2:0
\ Timer will use interrupt priority channel 2
IOBASE 0268 + CONSTANT TMRD2_GPR  EEWORD
0007 CONSTANT TMRD2_PLR_MASK  EEWORD
0003 CONSTANT TMRD2_PLR_PRIORITY  EEWORD \ pri'ty channel 2 in bits 2:0

\ Initialize Timer D2
: START-TMRD2

    \ Set compare 1 register to desired # of cycles
    TMRD2_COUNT TMRD2_CMP1 !

    \ Set reload register to zero
    0 TMRD2_LOAD !

    \ Timer control register
    \  001 = normal count mode
    \     1 011 = IPbus clock / 8 = 5 MHz timer clock
    \          0 0 = secondary count source n/a
    \             0 = count repeatedly
    \              1 = count until compare, then reinit
    \               0 = count up
    \                 0 = no co-channel init
    \                  000 = OFLAG n/a
    \  0011 0110 0010 0000 = $3620
    3620 TMRD2_CTRL !

    \ Timer status & control register
    \ Clear TCF flag, set interrupt enable flag
    8000 TMRD2_SCR CLEAR-BITS
    4000 TMRD2_SCR SET-BITS

    \ Interrupt Controller
    \ set the interrupt channel = 3 for Timer D3
    TMRD2_PLR_MASK     TMRD2_GPR CLEAR-BITS
    TMRD2_PLR_PRIORITY TMRD2_GPR SET-BITS
```

```
    \ enable that interrupt channel in processor status register
    GPIO_IPL_2 GPIO_IPR SET-BITS
;  EEWORD


\ Stop Timer D2
: STOP-TMRD2
    \ Timer control register
    \  000x xxxx xxxx xxxx = no count
    E000 TMRD2_CTRL CLEAR-BITS

    \ Timer status & control register
    \ Clear TCF flag, clear interrupt enable flag
    C000 TMRD2_SCR CLEAR-BITS
;  EEWORD

VARIABLE TICKS  EEWORD

\ High level word to handle the timer D2 interrupt
: TMRD2-IRPT
   \ clear the TCF flag to clear the interrupt
    8000 TMRD2_SCR CLEAR-BITS
    \ increment the ticks counter
    1 TICKS +!
;  EEWORD


HEX 0041 CONSTANT WP  EEWORD

CODE-SUB INT-SERVICE
DE0B P,           \  LEA  (SP)+
D00B P,           \  MOVE X0,X:(SP)+
D10B P,           \  MOVE Y0,X:(SP)+
D30B P,           \  MOVE Y1,X:(SP)+
D08B P,           \  MOVE A0,X:(SP)+
D60B P,           \  MOVE A1,X:(SP)+
D28B P,           \  MOVE A2,X:(SP)+
D18B P,           \  MOVE B0,X:(SP)+
D70B P,           \  MOVE B1,X:(SP)+
D38B P,           \  MOVE B2,X:(SP)+
D80B P,           \  MOVE R0,X:(SP)+
D90B P,           \  MOVE R1,X:(SP)+
DA0B P,           \  MOVE R2,X:(SP)+
DB0B P,           \  MOVE R3,X:(SP)+
DD0B P,           \  MOVE N,X:(SP)+
DE8B P,           \  MOVE LC,X:(SP)+
DF8B P,           \  MOVE LA,X:(SP)+
F854 P, OBJREF P, \  MOVE X:OBJREF,R0
FA54 P, WP P,     \  MOVE X:WP,R2
D80B P,           \  MOVE R0,X:(SP)+
DA1F P,           \  MOVE R2,X:(SP)   ; Note no increment on last push!
87D0 P, ' TMRD2-IRPT CFA P,   \  MOVE #$XXXX,R0   ; CFA of the word to
execute
E9C8 P, ATO4 P,   \  JSR  ATO4        ; do that Forth word
FA1B P,           \  MOVE X:(SP)-,R2  ; restore the saved wp
F81B P,           \  MOVE X:(SP)-,R0  ; restore the saved objref
```

```
FF9B P,            \  MOVE X:(SP)-,LA
DA54 P, WP P,      \  MOVE R2,X:WP
D854 P, OBJREF P, \  MOVE R0,X:OBJREF
FE9B P,            \  MOVE X:(SP)-,LC
FD1B P,            \  MOVE X:(SP)-,N
FB1B P,            \  MOVE X:(SP)-,R3
FA1B P,            \  MOVE X:(SP)-,R2
F91B P,            \  MOVE X:(SP)-,R1
F81B P,            \  MOVE X:(SP)-,R0
F39B P,            \  MOVE X:(SP)-,B2
F71B P,            \  MOVE X:(SP)-,B1
F19B P,            \  MOVE X:(SP)-,B0
F29B P,            \  MOVE X:(SP)-,A2
F61B P,            \  MOVE X:(SP)-,A1
F09B P,            \  MOVE X:(SP)-,A0
F31B P,            \  MOVE X:(SP)-,Y1
F11B P,            \  MOVE X:(SP)-,Y0
F01B P,            \  MOVE X:(SP)-,X0
EDD9 P,            \  RTI
END-CODE  EEWORD

\ Install the interrupt vector in Program Flash ROM
E984                    TMRD2_VECTOR PF!       \ JMP instruction
' INT-SERVICE CFA 2+   TMRD2_VECTOR 1+ PF!    \ target address
```

To install this interrupt you must have an IsoMax kernel version 0.5 or greater. This has a table of two-cell interrupt vectors starting at $7D80. The first cell (at $7D80+$40 for Timer D2) must be a machine-code jump instruction, $E984; the second cell is the address of the interrupt service routine. This address is obtained with the phrase `' INT-SERVICE CFA 2+` because the first two locations of a CODE-SUB or CODE-INT are "overhead." The interrupt vector is not installed with `EEWORD`; instead, it is programmed directly into Program Flash ROM with the `PF!` operator.

Observe also the use of `' TMRD2-IRPT CFA` to obtain the address "xxxx" of the high-level interrupt service routine.

This example is shown running out of Program ROM; that is, the words have been committed to Flash ROM with `EEWORD`. In an application you want your interrupt handler to reside in ROM so that it survives a reset or a memory crash. (Leaving an interrupt vector pointing to RAM, and then power-cycling the board, can cause the board to lock up.)

1.16. Harvard Memory Model

The IsoPod Processor uses a "Harvard" memory model, which means that it has separate memories for Program and Data storage. Each of these memory spaces uses a 16-bit address, so there can be 64K 16-bit words of Program ("P") memory, and 64K 16-bit words of Data ("X") memory.

1.16.1. MEMORY OPERATORS

Most applications need to manipulate data, so the memory operators use Data space. These include

```
@   !   C@   C!   +!   HERE   ALLOT   ,   C,
```

Occasionally you will need to manipulate Program memory. This is accomplished through a separate set of memory operators having a "P" prefix:

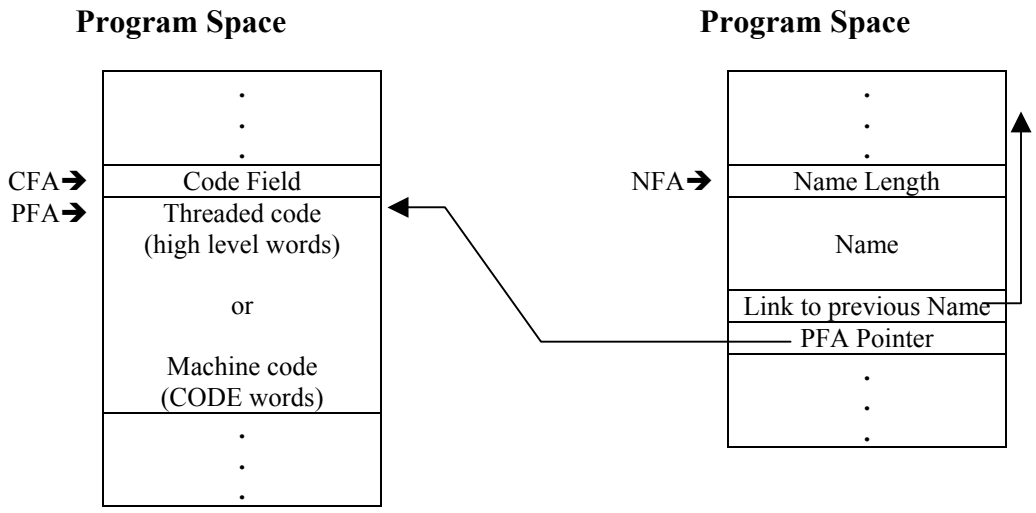
```
P@   P!   PC@   PC!   PHERE   PALLOT   P,   PC,
```

Note that on the IsoPod™, the smallest addressable unit of memory is one 16-bit word. This is the unpacked character size. This is also the "cell" size used for arithmetic and addressing. Therefore, `@` and `C@` are equivalent, and `!` and `C!` are equivalent.

1.16.2. WORD STRUCTURE

The executable "body" of a IsoMax™ word is kept in Program space. This includes the Code Field of the word, and the threaded definition of high-level words or the machine code definition of CODE words.

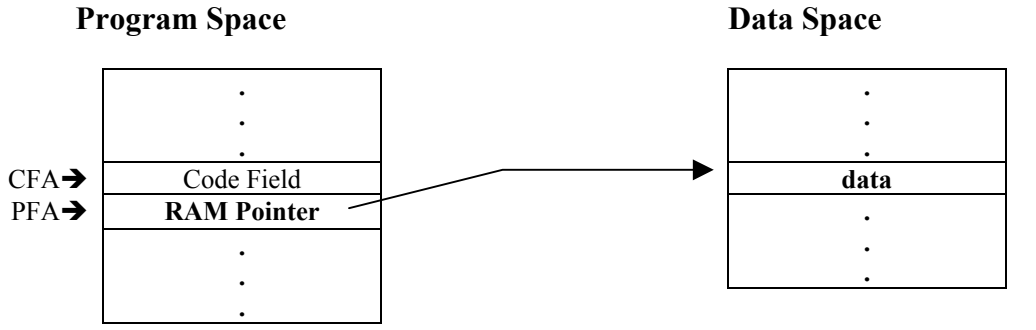
The "header" of a IsoMax™ word is also kept in Program space. This includes the Name Field, the Link Field, and the PFA Pointer. However, this may be stored separately from the executable “body.” This is to allow the headers, which aren’t used for an embedded application, to be easily stripped.



If you have not enabled separated heads, the words you add to the dictionary will have the header immedately before the executable body.

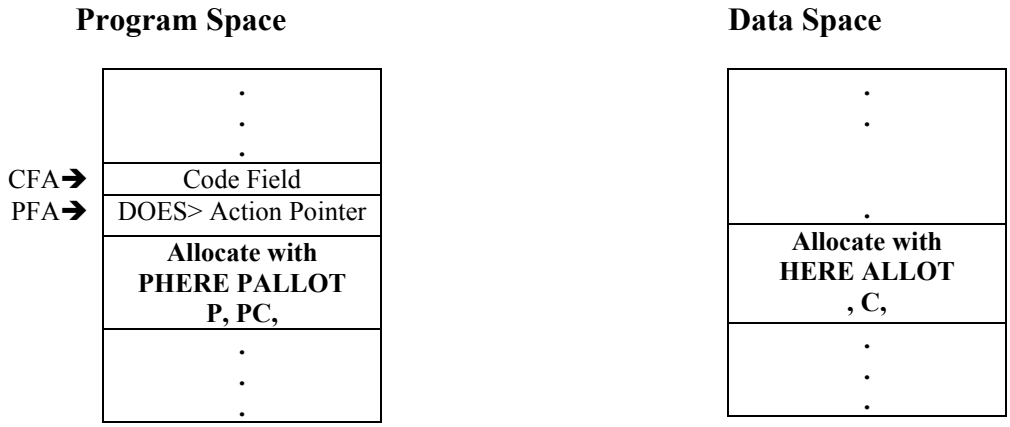
1.16.3. VARIABLES

Since the Program space is normally ROM, and variables must reside in RAM and in Data space, the "body" of a VARIABLE definition does not contain the data. Instead, it holds a pointer to a RAM location where the data is stored.



1.16.4. <BUILDS DOES>

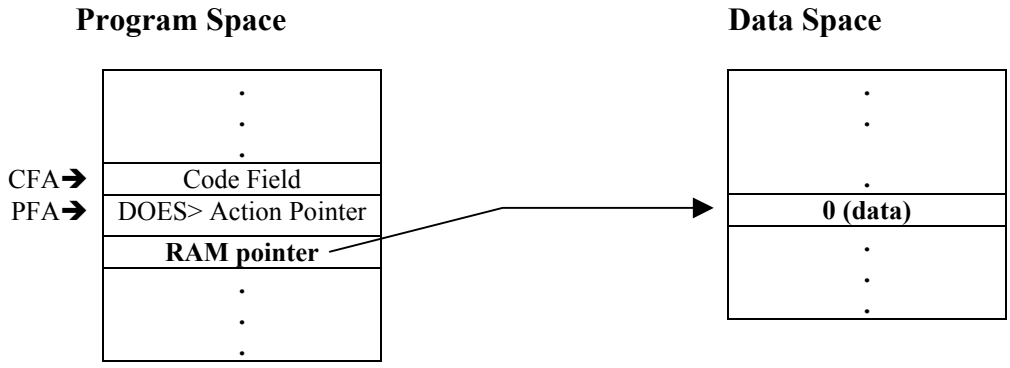
"Defining words" created with <BUILDS and DOES> may have a variety of purposes. Sometimes they are used to build Data objects in RAM, and sometimes they are used to build objects in ROM (i.e., in Program space). In the <BUILDS code you can allocate either space by using the appropriate memory operators.



For maximum flexibility, DOES> will leave on the stack the address *in Program space* of the user-allocated data. If you need to allocate data in Data space, you must also store (in Program space) a pointer to that data. For example, here is how you might define VARIABLE using <BUILDS and DOES>.

```
: VARIABLE
  <BUILDS     Defines a new Forth word, header and empty body;
    HERE      gets the address in Data space (HERE) and appends that to Program space;
    0  ,      appends a zero cell to Data space.
  DOES>       The "run-time" action will start with the Program address on the stack;
    P@        fetch the cell stored at that address (a pointer to Data) and return that.
;
```

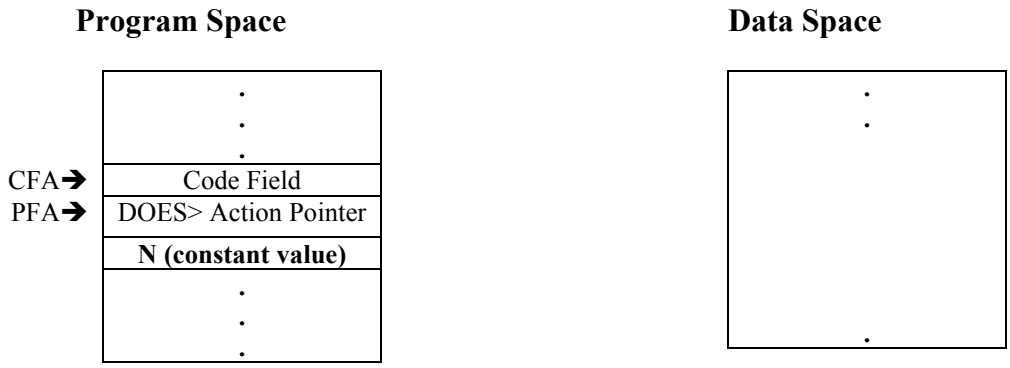
This constructs the following:



Words with constant data, on the other hand, can be allocated entirely in Program space. Here's how you might define CONSTANT:

```
: CONSTANT  ( n -- )
  <BUILDS     Defines a new Forth word, header and empty body;
    P,        appends the constant value (n) to Program space.
  DOES>       The "run-time" action will start with the Program address on the stack;
    P@        fetch the cell stored at that address (the constant) and return that.
;
```

This constructs the following:



1.17. Object Oriented Internals

For this illustration we will use the **BYTEIO** class from the file Gpioobj.4th (appended below).

1.17.1. Dictionary Hiding

BEGIN-CLASS marks the start of definitions that will be "hidden." Once they are hidden, they will only be visible to members of this class. **BEGIN-CLASS** just marks a dictionary position; it doesn't compile anything.

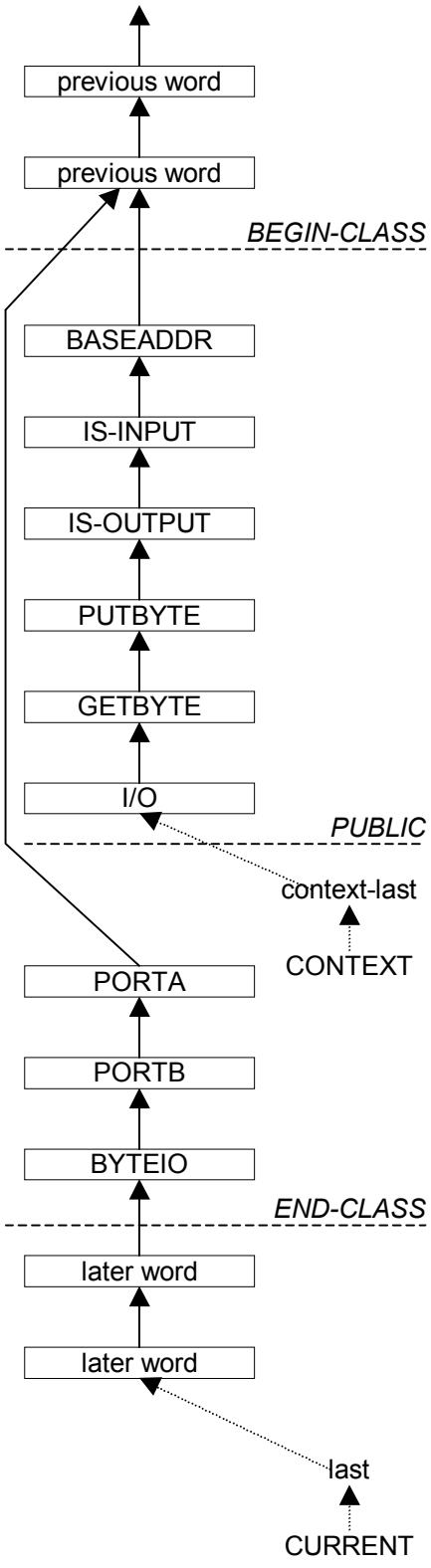
PUBLIC marks the end of the hidden definitions. It does two things. First, it puts a pointer to the last-defined word (i.e., the last hidden word) in the **context-last** variable. This means these words will still be found when the **CONTEXT** list is searched. Second, it relinks the main dictionary list around the hidden words, by resetting the **last** variable.

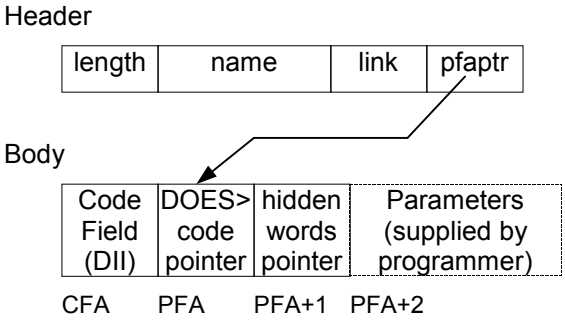
At this point, the hidden words are still searchable, and can still be used to write Forth definitions. New definitions will be "public" and will be part of the main dictionary list, not the hidden list.

END-CLASS hides the private definitions, by clearing **CONTEXT**. It also creates a class-name word (in this example, **BYTEIO**) which will make the private word list visible again, by putting its dictionary link back into the **context-last** variable.

1.17.2. Object Action

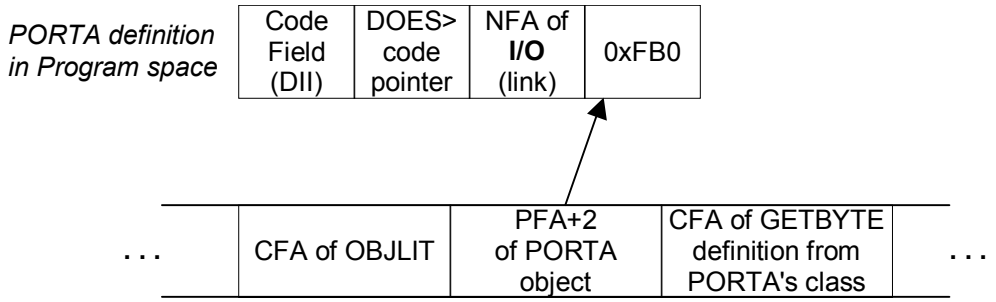
A word created with **OBJECT** has both a compile-time action and a run-time action. At compile-time (or when interpreted), it makes its hidden word list visible, by putting the dictionary link into the **context-last** variable. Thus, after an object is named, its private "methods" can be compiled or interpreted.





At run-time, an object puts the address of its parameters (PFA+2) into the **OBJREF** variable. This is essentially the same as **DOES>**, except that the address is stored into a variable instead of being left on the stack. The "methods" which follow the object all expect to find this address in **OBJREF**. (The word **SELF** returns this address.)

Note: when an object is used in a Forth definition, what actually gets compiled is a *literal* (in-line constant) with the address PFA+2. Thus the phrase **PORTA GETBYTE** is compiled as



The special word **OBJLIT** takes the in-line value which follows, and stores it in the **OBJREF** variable. This is exactly the same as the Forth primitive **LIT**, except that the value is stored in a variable instead of being left on the stack.

In this example, the **PORTA** definition has one user-supplied parameter: the value 0xFB0, which is the I/O address of the desired port. The object is created, and this extra parameter is appended, by the word **I/O** (see below).

```
\ ------------------------------------------------------
\ GPIO PARALLEL PORTS - BYTE I/O
\ ------------------------------------------------------
BEGIN-CLASS BYTEIO

\ BYTEIO methods expect SELF to point to:  baseaddr  in ROM
: BASEADDR ( -- a )   SELF P@ ;

: IS-INPUT    ( makes pin an input
    0FF  BASEADDR 3 + CLEAR-BITS   ( PER=0, GPIO
    0FF  BASEADDR 2+  CLEAR-BITS   ( data dir=in
```

```
;

: IS-OUTPUT   ( makes pin an output
    0FF  BASEADDR 3 + CLEAR-BITS   ( PER=0, GPIO
    0FF  BASEADDR 2+  SET-BITS     ( data dir=out
;

: PUTBYTE ( c -- )   IS-OUTPUT  BASEADDR 1+ C! ;
: GETBYTE ( -- c )   IS-INPUT   BASEADDR 1+ C@ ;

\ define an I/O port
: I/O ( baseaddr -- )   OBJECT  P, ;

PUBLIC

FB0 I/O PORTA
FC0 I/O PORTB

END-CLASS BYTEIO
```

1.18. CPU Registers

Under construction…

```
 ( BASE REGISTERS)
0C00 SIM
0C40 PFIU2
0D00 TMRA
0D20 TMRB
0D40 TMRC
0D60 TMRD
0D80 CAN
0E00 PWMA
0E20 PWMB
0E40 DEC0
0E50 DEC1
0E60 ITCN
0E80 ADCA
0EC0 ADCB
0F00 SCI0
0F10 SCI1
0F20 SPI
0F30 COP
0F40 PFIU
0F60 DFIU
0F80 BFIU
0FA0 CLKGEN
0FB0 GPIOA
0FC0 GPIOB
0FE0 GPIOD
0FF0 GPIOE

( TIMER REGISTERS. OFFSET IS CHANNEL  * 8 )


0 CMP1
1 CMP2
2 CAP
3 LOAD
4 HOLD
5 CNTR
6 CTRL
7 SCR

( GPIO )

0 PUR
1 DR
2 DDR
3 PER
4 IAR
5 IENR
6 IPOLR
```

7 IPR
8 IESR

(A/D CONVERTER)

0 ADCR1
1 ADCR2
2 ADZCC
3 ADLST1
4 ADLST2
5 ADSDIS
6 ADSTAT
7 ADLSTAT
8 ADZCSTAT
9 ADRSLT0
A ADRSLT1
B ADRSLT2
C ADRSLT3
D ADRSLT4
E ADRSLT5
F ADRSLT6
10 ADRSLT7
11 ADLLMT0
12 ADLLMT1
13 ADLLMT2
14 ADLLMT3
15 ADLLMT4
16 ADLLMT5
17 ADLLMT6
18 ADLLMT7
19 ADHLMT0
1A ADHLMT1
1B ADHLMT2
1C ADHLMT3
1D ADHLMT4
1E ADHLMT5
1F ADHLMT6
20 ADHLMT7
21 ADOFS0
22 ADOFS1
23 ADOFS2
24 ADOFS3
25 ADOFS4
26 ADOFS5
27 ADOFS6
28 ADOFS7

(PWM)

0 PMCTL
1 PMFCTL
2 PMFSA
3 PMOUT
4 PMCNT
5 PWMCM
6 PWMVAL0

7 PWMVAL1
8 PWMVAL2
9 PWMVAL3
A PWMVAL4
B PWMVAL5
C PMDEADTM
D PMDISMAP1
E PMDISMAP2
F PMCFG
10 PMCCR
11 PMPORT

(QUAD)

0 DECCR
1 FIR
2 WTR
3 POSD
4 POSDH
5 REV
6 REVH
7 UPOS
8 LPOS
9 UPOSH
A LPOSH
B UIR
C LIR
D IMR
E TSTREG

(SCI)

0 SCIBR
1 SCICR
2 SCISR
3 SCIDR

(SPI)

0 SPSCR
1 SPDSR
2 SPDRR
3 SPDTR