Colour Maximite 2

Threads User Manual

Threads in MMBasic programs on the Colour Maximite 2

For firmware 8.00 with threads (branch threads)

Peter Mather

(26 November 2026)

The firmware and this manual are at
https://github.com/UKTailwind/CMM2V1.5

About This Guide

This manual is for MMBasic programmers who want a program to do several things at once with threads: SUBs that run alongside the main program, started with THREAD START. It covers starting and ending threads, how they take turns, which commands let the others run while they wait, sharing data safely with locks (mutexes), holds and flags, interrupts and the network with threads, errors and debugging, memory and limits, and complete programs. It assumes a working knowledge of MMBasic; the user manual has the same commands in its reference sections and an overview in its Appendix R.

Support

Questions should be raised on the Back Shed forum (http://www.thebackshed.com/forum/Microcontrollers), and problems with threads reported as issues of the repository above.

Copyright

This manual is copyright 2026 Peter Mather and is distributed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Australia license (CC BY-NC-SA 3.0), as the user manual is.

Contents

Introduction 5
What threads are 5
Why use them 5
A first program 5
How this manual is arranged 6
Starting and Ending Threads 7
THREAD START 7
Arguments 7
SIZE, LOCALS and NEST 7
How a thread ends 7
The end of the program 8
PAUSE and RESUME 8
Is it running? 8
How Threads Run 9
Taking turns 9
Priority 9
Statements are never split 9

Waits that let the other threads run 10
Waits that hold every thread 10
The cost 10
OPTION COMPILE 10
Sharing Data Safely 11
What each thread has of its own 11
What they share 11
The danger: two statements that belong together 11
Way 1: one statement 11
Way 2: one writer, one owner 11
Way 3: a lock 12
Way 4: THREAD HOLD 14
Passing messages with flags 14
Waiting for a thread 14
The screen and the keyboard 14
The usual names 15
Interrupts and Threads 16
An interrupt belongs to the thread that set it up 16
The pattern: set up, then wait 16
When the interrupt waits 16
When its thread ends 16
Inside an interrupt routine 16
The Network and Bluetooth 18
The network belongs to one thread 18
Waiting on the network 18
Bluetooth 18
Errors and Debugging 19
An error ends the program 19
The list of threads 19
THREAD LIST 19
TRACE 19
MEMORY 19
OPTION PROFILING 19
Stack overflow 19
Common mistakes 20
Memory and Limits 21
What a thread takes 21
The stack 21
Local variables 21
Loops and calls (NEST) 21
Limits 21
What cannot be done in a thread 22
Things to know 22
Worked Programs 23
Balls and a clock while the user types 23
A long job with a live screen 23

Workers sharing a total 24
A sensor logger 24
A web request while a game runs 25
Appendix A – The Commands in Brief 26
The error messages 26

Introduction

What threads are

A thread is a SUB of your program that runs alongside the main program. Once started, it goes on by itself: it can loop, wait, call other SUBs and FUNCTIONs, and keep its own LOCAL variables, while the main program carries on with its own work. A program can have up to sixteen threads at once.

The Colour Maximite 2 has one processor, so only one thread runs at any instant. The firmware shares the processor between them: each runs for a short time slice, a millisecond unless you choose otherwise, and the next one then has its turn. The switch from one thread to the next always comes between two statements, never in the middle of one. Because the slices are short, the threads appear to run at the same time.

Why use them

MMBasic programs have long done several things at once with interrupts: SETTICK calls a routine every so often, SETPIN calls one when a pin changes, and the main program does the rest. That works well when each event needs a short, quick answer. It works less well when the job has a flow of its own: wait a second, do something, wait for a key, do something else. Written with interrupts, such a job becomes a routine that is called again and again and a variable that remembers where it had got to.

A thread lets you write that job as the straight sequence of steps it is:

```
Sub TrafficLights
  Do
    Lamp RGB(red)    : Pause 5000
    Lamp RGB(yellow) : Pause 1000
    Lamp RGB(green)  : Pause 4000
    Lamp RGB(yellow) : Pause 1000
  Loop
End Sub
```

While this thread is in a PAUSE it uses no processor time, and the rest of the program runs at full speed.

Threads suit:

- something with a rhythm of its own: a clock in the corner of the screen, a status line, a flashing LED, a tune played note by note;
- many of the same thing: each ball, alien or robot of a game as the same SUB in a thread of its own, with its own variables;
- a long calculation with a live screen: the work in a thread, the main program showing progress and reading the keys;
- waiting on the outside world: a thread waits on the network, a serial port or a sensor while the rest of the program goes on;
- a program that listens while it works: the main program sits in INPUT for the next command while threads run the machine or the display.

A first program

```
Option Explicit
Dim Integer beats

Sub Heart
  Do
    Inc beats
    Pause 500
  Loop
End Sub

Thread Start 1, Heart
Do
  Text 0, 0, "Beats: " + Str$(beats) + "  "
  Pause 100
Loop Until Inkey$ <> ""
End
```

THREAD START 1, Heart starts thread 1 running the SUB Heart. From then on two things go on at once: Heart adds one to beats every half a second, and the main program shows beats ten times a second until a key is pressed. END stops the program, and its threads with it.

How this manual is arranged

- **Starting and Ending Threads**: THREAD START, its arguments, and the ways a thread ends.
- **How Threads Run**: the time slice, priorities, which commands let the other threads run while they wait, and what threads cost.
- **Sharing Data Safely**: what each thread has of its own, what they share, and the locks (the mutex of other systems), holds and flags that keep shared data right.
- **Interrupts and Threads**: interrupts belong to the thread that sets them up.
- **The Network and Bluetooth**: each belongs to one thread.
- **Errors and Debugging**: what happens at an error, Ctrl-C and END, and THREAD LIST, TRACE, MEMORY and profiling.
- **Memory and Limits**: what a thread takes, SIZE, LOCALS and NEST, and what cannot be done in a thread.
- **Worked Programs**: complete programs that use threads.
- **Appendix A**: the commands and functions in brief.

The firmware's user manual has the same commands in its reference sections, and an overview in its Appendix R.

Starting and Ending Threads

THREAD START

```
THREAD START n, sub [, argument ...]
THREAD START n SIZE kb LOCALS m NEST d PRIORITY p, sub [, argument ...]
```

THREAD START starts thread 'n', a number from 1 to 16 that you choose, running the SUB 'sub'. The main program is thread 0. THREAD START returns at once: the new thread runs when its turn comes, a moment later.

The SUB must be a SUB of the program (not a FUNCTION, and not a CSUB), and thread 'n' must not already be running ("Thread n is running"). The number becomes free again when the thread ends. THREAD START is for programs: at the command prompt it is refused.

A thread starts with the print position, colours, font and write page of the thread that started it.

Arguments

Up to sixteen arguments may follow the SUB's name, as in an ordinary call:

```
Sub Blink(pin%, ms%)
  SetPin pin%, DOUT
  Do
    Pin(pin%) = 1 - Pin(pin%)
    Pause ms%
  Loop
End Sub

Thread Start 1, Blink, 7, 250
Thread Start 2, Blink, 8, 400
```

Each argument is worked out by THREAD START, in the thread that runs it, and handed to the new thread's SUB by value: the new thread has its own copy, and changing its parameter changes nothing outside it. Integers, floats and strings are all passed exactly, and the usual conversions apply (an integer for a float parameter, and so on).

A whole array, written as its name with empty brackets, is passed as itself, so that the thread can fill it or read it:

```
Dim Float samples(99)
Thread Start 3, Recorder, samples(), 10
```

It must be a global array. A SUB's own array would vanish when that SUB returned while the thread still used it, so it is refused ("Only a global array can be passed to a thread").

Several threads may run the same SUB at the same time, each with its own parameters and LOCAL variables. MM.INFO(THREAD) tells a SUB which thread is running it.

SIZE, LOCALS and NEST

Each thread has a stack of its own, for the SUB and FUNCTION calls it makes, room for a number of LOCAL variables, and its own lists of the FOR loops, DO loops and calls it is inside. Unless you say otherwise a thread gets a 20 KB stack, 48 local variables, and loops and calls 20 deep. SIZE, LOCALS and NEST, after the thread's number and in any order, set them for that thread:

```
Thread Start 4 Size 12 Locals 8 Nest 4, Clock   ' a small thread
Thread Start 5 Size 48 Locals 100, Solver       ' a deep recursion
```

SIZE is 12 to 512 KB, LOCALS 8 to 256 and NEST 4 to 100. THREAD SIZE sets the defaults for threads started later without their own:

```
Thread Size 12, 16                              ' stack 12 KB, 16 locals
Thread Size 12, 16, 8                           ' and loops and calls 8 deep
```

They go back to 20 KB, 48 and 20 at the next RUN. PRIORITY p, in the same place, gives the thread more of the processor than the others: see *Priority* in the next chapter. The chapter *Memory and Limits* says how to choose them.

How a thread ends

A thread ends:

- when its SUB ends (END SUB, or EXIT SUB at its top level);

- at THREAD EXIT, wherever it is, even deep inside SUBs and FUNCTIONs;
- when another thread stops it with THREAD STOP n;
- when the program ends.

When a thread ends, its LOCAL variables and the memory it used are freed, its locks are given up, and if the network or Bluetooth belonged to it they are free again. Its number may then be used for a new thread.

THREAD STOP n ends thread 'n' wherever it is: in a loop, in a PAUSE, inside a FUNCTION. Nothing happens if thread 'n' is not running. A thread may stop itself, which is the same as THREAD EXIT.

THREAD EXIT is an error in the main program ("Not in a thread"): the main program ends with END.

The end of the program

The program ends when any thread, the main program included, runs END, when an error is not caught by ON ERROR, or at Ctrl-C. Every thread stops with it, wherever it is. SUB MM.END, if the program has one, runs in the thread that ran END.

The program also ends when the main program runs off its end. If the main program should wait for its threads to finish first, THREAD WAIT does it:

```
Thread Start 1, Worker
Thread Start 2, Worker
Thread Wait 1
Thread Wait 2
Print "Both done"
```

PAUSE and RESUME

THREAD PAUSE n holds thread 'n': it stays where it is and does not run until THREAD RESUME n. Its interrupts wait for it too. A thread may pause itself, and stops at the end of that statement; another thread then has to resume it. Both are errors if thread 'n' is not running ("Thread n is not running").

```
If k$ = "P" Then Thread Pause 1
If k$ = "R" Then Thread Resume 1
```

Is it running?

MM.INFO(THREAD n) gives the state of thread 'n':

Value	Thread n is
0	not running: never started, or ended
1	running, or ready to run when its turn comes
2	in a PAUSE
3	paused by THREAD PAUSE
4	waiting: for a lock, another thread, a flag, the next frame or the network

MM.INFO(THREAD 0), the main program, is always 1. MM.INFO(THREAD), with no number, is the number of the thread that runs it.

How Threads Run

Taking turns

Only one thread runs at a time. Each runs until one of these happens, and then the next thread that can run has its turn:

- its time slice is up: 1 ms unless THREAD SLICE says otherwise;
- it waits: PAUSE, INPUT, THREAD LOCK, THREAD WAIT and the other waits of the next section;
- it runs THREAD YIELD, which gives up the rest of its slice;
- an interrupt that belongs to another thread is due (see *Interrupts and Threads*).

The threads take turns in number order, the main program among them as thread 0. A thread whose PAUSE has just ended goes first, ahead of the others, so that a thread that does something every 50 ms does it on time even while others are busy.

THREAD SLICE ms sets the slice, from 1 to 100 ms. A longer slice means fewer switches, so slightly less time lost to them, but a busy thread then keeps the others waiting longer. One millisecond suits almost every program.

Priority

Every thread, the main program included, has a priority from 1 to 10. It is 1 unless you say otherwise. A thread's priority is the number of time slices in its turn: a thread at priority 5 beside one at priority 1 runs for five slices, then the other runs for one, and round again. So two busy threads at 5 and 1 share the processor five to one.

```
Thread Start 1 Priority 5, Compute        ' five slices a turn
Thread Start 2, Logger                    ' one
Thread Priority 0, 3                      ' the main program: three
Thread Priority 2, 4                      ' thread 2 from now on: four
```

THREAD PRIORITY n, p changes the priority of a running thread (n = 0 is the main program) from its next turn. All priorities are 1 again at the next RUN, and THREAD LIST shows any that is not 1.

- **No thread is left out.** The threads still take turns in number order: a higher priority only makes a turn longer. A thread waits for its turn while each of the others has its own, whatever its own priority is.
- **Priority shares the time of busy threads.** A thread that waits (PAUSE, INPUT, THREAD LOCK and the rest) gives up what is left of its turn, as it always did, and a thread whose PAUSE has just ended still goes first. So a thread that mostly waits needs no priority to be on time.
- **Long turns make the others wait.** With one busy thread at priority 10, the others run once every 10 ms instead of every 1 ms. A key typed into INPUT is then echoed about 5 ms later than usual, and beside three such threads about 15 ms later. THREAD SLICE multiplies this, so keep the slice at 1 ms when you use high priorities.
- **A lock lends priority.** While a thread waits at THREAD LOCK, the thread that holds the lock has the waiting thread's turn, if that is longer than its own, until it unlocks. So a thread at priority 1 that holds a lock does not keep a thread at priority 9 waiting at its own slow pace. THREAD WAIT and the flags lend nothing.

Statements are never split

The switch to another thread is only ever made between two statements. A statement is always whole: no other thread can see it half done. This is what makes threads in MMBasic straightforward to use:

- INC total, or total = total + 1, in two threads never loses a count;
- a PRINT always prints all its items together;
- a LINE, BOX or TEXT is always drawn whole.

The other side is that a long statement holds every thread until it ends. A SORT of a large array, a COPY of a big file, a LOAD PNG or a PIXEL FILL of the whole screen are each one statement: the other threads wait for them.

One exception to "whole": a FUNCTION of your own, called from inside a statement, is made of statements, and the switch can come between them. The statement that called it continues afterwards, so the statement still happens, but other threads may run before it finishes.

Waits that let the other threads run

A thread that waits uses no processor time: the others run until what it waits for has happened. These commands wait in this way:

Command	Waits for
PAUSE, CPU SLEEP	the time
INPUT, LINE INPUT	a line typed at the keyboard or the console, or from a COM port
FRAMEBUFFER WAIT, PAGE COPY ...,B, PAGE DISPLAY, FRAMEBUFFER WINDOW ...,B, MAP SET, RESET, GREYSCALE and MAXIMITE	the start of the next frame
SYNC	the end of SYNC's period
ADC START (with no interrupt)	the samples
TEMPR()	the sensor's conversion
WEB commands (requests, NTP, MQTT, PING, UDP)	the network's answer
BLUETOOTH commands	the device
THREAD LOCK	the lock
THREAD WAIT	a thread to end, or a flag

So a program can sit in INPUT for its user while threads keep the screen moving, or wait two seconds for a web server while a game goes on.

When two threads wait for the same frame (FRAMEBUFFER WAIT, PAGE COPY ...,B), both get it: each wakes at the start of the next frame.

Waits that hold every thread

Other commands that take time hold every thread until they end, as any long statement does. Among them:

- the timed pin commands: PULSIN(), DISTANCE(), BITSTREAM, DEVICE SERIALRX, the DHT22 and 1-Wire timing, which must not be interrupted;
- the SD card, USB drive, I2C and SPI transfers, which take milliseconds;
- PRINT to the console or a COM port whose output buffer is full, until there is room;
- PLAY BBC SOUND with its queue full.

The cost

A program that starts no thread runs exactly as fast as it did before threads existed. With threads, a switch from one thread to another takes about 5 microseconds, so 1 ms slices cost about half a percent of the processor. A thread in a wait costs nothing.

Starting a thread takes about a third of a millisecond, mostly for preparing its stack.

OPTION COMPILE

Threads work with OPTION COMPILE ON and SHADOW as they do without: the compiled statements of several threads, even of the same SUB at the same time, each work on their own thread's variables.

Sharing Data Safely

What each thread has of its own

Each thread has its own:

- LOCAL variables and SUB and FUNCTION parameters;
- FOR, DO and WHILE loops, and the SUBs, FUNCTIONs and GOSUBs it is inside;
- ON ERROR setting, MM.ERRNO and MM.ERRMSG$;
- place in the DATA statements for READ (each thread reads from the start, or from its own RESTORE);
- EXECUTE line;
- print position, colours, font, PRINT @ mode and write page (PAGE WRITE, FRAMEBUFFER WRITE).

The last group means that one thread's COLOUR, FONT, PRINT @ or PAGE WRITE does not change another thread's. A new thread starts with those of the thread that started it, and MODE gives every thread the new mode's.

What they share

Everything else belongs to the whole program:

- global variables and arrays, and STATIC variables;
- the screen itself, its pages and the framebuffers;
- open files, the current drive and directory, and DIR$() going on from its last search;
- the keyboard, the mouse and the gamepads;
- sound, sprites, BLIT buffers, the turtle and the GUI's controls;
- the pins, I2C, SPI, 1-Wire and the COM ports, and MM.I2C and MM.ONEWIRE after them;
- the OPTIONs, TIMER, RANDOMIZE's sequence, SYNC's period and FLAG().

The danger: two statements that belong together

Because a switch can come between any two statements, a job that takes two statements can be cut in the middle by another thread doing the same job. Suppose two threads each add to a shared total like this:

```
v = total
total = v + 1
```

If a switch comes between the two lines, both threads read the same total, both add one, and one count is lost. The same thing happens with:

- `If free > 0 Then free = free - 1` (the test and the change);
- `Colour RGB(red)` then `Print "Alarm!"` in one thread, `Colour RGB(green)` and `Print` in another, *if* they share the print position;
- reading a record from a file in one thread while another moves the file's position.

These faults are timing faults: a program may run correctly a thousand times and fail the next. It is worth arranging a program so that they cannot happen. There are four ways.

Way 1: one statement

A job done in one statement cannot be cut. INC total is one statement; so is total = total + 1. TEXT with its colour (TEXT x, y, s$, , , , RGB(red)) carries the colour with it.

Way 2: one writer, one owner

A variable that only one thread sets, and the others only read, is always safe. More generally, give each shared thing an owner: one thread reads the keyboard, one thread writes the log file, one thread draws each part of the screen. The other threads leave values in global variables for the owner, and the owner acts on them.

Way 3: a lock

When several threads must do the same multi-statement job on shared data, a lock lets only one of them do it at a time:

```
Thread Lock 1
v = total
total = v + 1
Thread Unlock 1
```

Every thread that does the job takes the same lock first. A thread that reaches THREAD LOCK 1 while another holds it waits there, using no processor time, until the holder reaches THREAD UNLOCK 1. There are sixteen locks, numbered 1 to 16: use a different number for each separate thing that needs protecting.

A lock is what other languages and operating systems call a *mutex*. The sections after *Deadlock* say how to program with one.

The rules:

- A lock belongs to the thread that took it. Taking it again in the same thread is an error ("Lock k is already held by this thread"), and so is unlocking a lock another thread holds ("Lock k is not held by this thread").
- A thread that ends, or is stopped, gives up its locks.
- THREAD LOCK k, timeout waits at most 'timeout' milliseconds; if the lock is still taken then, it is the error "Lock k timed out". A program that wants to go on without the lock catches the error with ON ERROR SKIP.
- MM.INFO(THREAD LOCK k) is the thread that holds lock k (0 the main program), or -1.
- Keep the statements between LOCK and UNLOCK few and quick: other threads that want the lock wait meanwhile.
- While a thread of a higher priority waits for a lock, the thread that holds it has that thread's longer turn until it unlocks (see *Priority* in *How Threads Run*), so a thread of a low priority does not keep it waiting long.

Deadlock

Two threads that each hold one lock and wait for the other's wait for ever:

```
' thread 1                  ' thread 2
Thread Lock 1               Thread Lock 2
Thread Lock 2               Thread Lock 1
```

Avoid it by always taking locks in the same order (1 before 2, everywhere). If a program does stop like this, Ctrl-C lists each thread with the lock it waits for and the thread that holds it.

A lock is a mutex

A *mutex*, short for mutual exclusion, is something that only one thread can have at a time. MMBasic has no command of that name because THREAD LOCK and THREAD UNLOCK are it: THREAD LOCK takes the mutex (other systems say acquire) and THREAD UNLOCK gives it back (release). Where another system has a program create a mutex, lock it, unlock it and at last destroy it, here the sixteen are there from RUN, all free. A program chooses a number for each thing it wants to protect. Nothing is created, and nothing needs tidying away.

Give each number a name with CONST. The program then says what each mutex guards, and two parts of it cannot choose the same number by accident:

```
Const mxTotal = 1, mxLog = 2

Thread Lock mxTotal
v = total
total = v + 1
Thread Unlock mxTotal
```

A mutex protects nothing by itself. It is an agreement: every part of the program that works on the shared thing takes the same mutex first, and a part that does not is not stopped. The sure way to keep the agreement is to put the lock, the work and the unlock in one SUB, and to have every thread do the work by calling it:

```
Sub Deposit(amount%)
  Local Integer v
  Thread Lock mxTotal
  v = total
```

```
  total = v + amount%
  Thread Unlock mxTotal
End Sub
```

Leave such a SUB only at its end. An EXIT SUB between the LOCK and the UNLOCK leaves the mutex taken: every other thread that wants it then waits for ever, and the same thread taking it again is the error "Lock k is already held by this thread".

A thread may wait while it has a mutex: PAUSE, INPUT, a file, the network. That is the difference from THREAD HOLD (*Way 4*, below), which stops every other thread and so allows no wait. With a mutex only the threads that want that mutex wait, and the rest run on. Use a mutex for longer work on one shared thing, such as a file that several threads write to or a display on an SPI bus, and a hold for two or three quick statements.

Trying a mutex

THREAD LOCK k, 0 takes the mutex if it is free and gives up at once if it is not. Giving up is the error "Lock k timed out", so ON ERROR SKIP goes before it, and MM.ERRNO says which happened:

```
On Error Skip
Thread Lock mxLog, 0
If MM.Errno = 0 Then
  Print #2, msg$                    ' it was free, and is this thread's now
  Thread Unlock mxLog
Else
  Inc missed                        ' another thread has it: do something else
EndIf
```

A larger number waits that long first: THREAD LOCK mxLog, 100 gives the thread that has it a tenth of a second to finish. MM.INFO(THREAD LOCK k) is for showing which thread has a mutex, not for deciding whether to take it: by the next statement another thread may have taken it.

A mutex the thread already has

A mutex is not counted: the thread that has it cannot take it a second time. That matters when a SUB that takes a mutex calls another SUB that takes the same one. The simplest cure is to decide which SUBs lock: the outer ones, which the threads call, take the mutex, and the inner ones they share expect to be called with it taken.

Where that is awkward, these two SUBs make a mutex that its owner may take again, which other systems call a recursive mutex. Each Enter is matched by a Leave, and the mutex is given back at the last Leave:

```
Dim Integer depth(16)

Sub Enter(k%)
  If MM.Info(Thread Lock k%) <> MM.Info(Thread) Then Thread Lock k%
  Inc depth(k%)
End Sub

Sub Leave(k%)
  Inc depth(k%), -1
  If depth(k%) = 0 Then Thread Unlock k%
End Sub
```

Only the thread that has the mutex changes its depth, so the count needs no protection of its own.

A mutex also belongs to its thread: only the thread that locked it can unlock it. So it cannot be locked in one thread and unlocked in another as a signal between them. The flags do that (*Passing messages with flags*, below).

Busy threads and a mutex

A thread that waits for a mutex gets it when its turn comes round and the mutex is free. A thread that unlocks and locks again straight away, in a loop with no wait in it, has usually locked it again by then. The others get in only when the end of its time slice falls between the UNLOCK and the LOCK. They do get in, but unevenly: of two such threads tried on a Generation 3, one did about 40% of the work and the other 60%. THREAD YIELD after the UNLOCK makes them take turns exactly:

```
Do
  Thread Lock mxTotal
  ' ... the work ...
  Thread Unlock mxTotal
  Thread Yield                      ' a thread that waits for it has it next
Loop
```

A loop with a PAUSE or any other wait outside the mutex needs no THREAD YIELD.

A mutex and interrupt routines

An interrupt routine runs in the thread that set it up, between two of that thread's statements. That can be between its THREAD LOCK and its THREAD UNLOCK, and between its THREAD HOLD and its THREAD RELEASE. So neither keeps a thread's own interrupt routine out: to the firmware the routine is that thread. In a routine:

- THREAD LOCK of a mutex its own thread has is the error "Lock k is already held by this thread";
- THREAD LOCK of one another thread has is the error "Cannot wait in an interrupt routine", since nothing else runs until the routine ends;
- THREAD LOCK of a free one takes it, and the routine must unlock it before it ends.

So keep mutexes out of interrupt routines. A routine may also come while another thread is half way through its protected work, and would see that work half done. Let the routine store what it has in a variable of its own and set a flag, and let its thread do the protected work when THREAD WAIT FLAG wakes it (see *Interrupts and Threads*).

Way 4: THREAD HOLD

THREAD HOLD stops every other thread from running until THREAD RELEASE. It is the simplest way to make two or three quick statements into one:

```
Thread Hold
If free > 0 Then free = free - 1 : got = 1
Thread Release
```

Use it only for a few quick statements: everything else waits, the other threads' interrupts included. A wait that cannot end at once (PAUSE, THREAD LOCK, INPUT and the rest) is an error inside a hold ("Cannot wait while the threads are held"), since no other thread could make it end. HOLD and RELEASE nest; RELEASE with no HOLD is an error.

Passing messages with flags

FLAG(0) to FLAG(63) are bits the whole program shares. THREAD WAIT FLAG f waits, using no processor time, until FLAG(f) is 1. This lets one thread wake another:

```
Sub Producer
  Do
    item = NewReading()
    Flag(1) = 1                    ' an item is ready
    Thread Wait Flag 2             ' until it has been taken
    Flag(2) = 0
  Loop
End Sub

Sub Consumer
  Do
    Thread Wait Flag 1
    Flag(1) = 0
    Process item
    Flag(2) = 1                    ' taken
  Loop
End Sub
```

THREAD WAIT FLAG does not clear the flag: the thread that waited clears it when it has dealt with it, as above. THREAD WAIT FLAG f, timeout gives up after 'timeout' milliseconds; the program then looks at FLAG(f) to see which happened.

An interrupt routine can set a flag too, which is how a thread waits for an event (see *Interrupts and Threads*).

Waiting for a thread

THREAD WAIT n waits until thread 'n' has ended, at once if it is not running. With a timeout it gives up after that many milliseconds, and MM.INFO(THREAD n) says whether the thread ended. A thread cannot wait for itself.

The screen and the keyboard

Each thread has its own print position, so two threads printing with PRINT @ or TEXT at different places do not disturb each other, and a thread drawing a clock does not move the line a user is typing at an INPUT. But the screen itself is one: two threads that PRINT without saying where both carry on from where they were and

write over each other, and CLS or a scroll moves what is under every thread's position. Give each thread its own part of the screen.

The keyboard is one too. INKEY$, INPUT and KEYDOWN in two threads at once race for the keys, each getting some. Let one thread read the keyboard, and pass what it reads to the others.

The usual names

For readers who know threads from another language or a real-time operating system, these are the names used there and what does the job here.

Elsewhere	**In MMBasic**
Mutex: create, lock (acquire, take), unlock (release, give)	THREAD LOCK k and THREAD UNLOCK k; the sixteen need no creating
Try-lock; a lock with a timeout	THREAD LOCK k, 0 and THREAD LOCK k, ms, each after ON ERROR SKIP
Recursive mutex	The Enter and Leave SUBs of *A mutex the thread already has*
Priority inheritance	Built in: a lock lends priority (*How Threads Run*)
Critical section; scheduler lock	THREAD HOLD and THREAD RELEASE
Atomic operation	Any single statement
Event, semaphore or condition variable	FLAG(f) with THREAD WAIT FLAG f
Join	THREAD WAIT n
Yield; sleep	THREAD YIELD; PAUSE
A thread's own storage	LOCAL variables and the SUB's parameters
The thread's identifier	MM.INFO(THREAD)

Interrupts and Threads

An interrupt belongs to the thread that set it up

SETTICK, SETPIN with an interrupt, ON KEY, a COM port's interrupt, the GUI's, the network's and all the others name a routine to call when something happens. With threads, the routine runs in the thread that set it up:

- a SETTICK set up by the main program, or before any thread was started, runs in the main program, as it always has;
- a SETTICK set up by thread 2 runs in thread 2.

The routine is its own context, as it always was: a SUB with its own LOCAL variables, that sees the global variables. It runs with its thread's print position, colours, font and write page, and it runs to its end, IRETURN or END SUB, before any other thread runs. Interrupt routines should still be short.

If the interrupt comes while its thread is not the one running, that thread runs at once to take it, ahead of the others.

The pattern: set up, then wait

A thread can set up an interrupt and then sleep until the routine says something has happened:

```
Dim Integer reading

Sub Sampler
  SetTick 100, Sample, 1
  Do
    Thread Wait Flag 1            ' asleep until the routine sets it
    Flag(1) = 0
    Text 0, 20, "Reading: " + Str$(reading) + "   "
  Loop
End Sub

Sub Sample
  reading = Pin(7)
  Flag(1) = 1
End Sub

Thread Start 1, Sampler
Input "Command"; c$               ' the sampler goes on meanwhile
```

The thread uses no processor time between readings; the main program can do anything, even sit in INPUT.

When the interrupt waits

An interrupt waits, and is not lost, while its thread cannot take it:

- **its thread is paused** (THREAD PAUSE): it runs when the thread is resumed;
- **its thread is in a wait that takes no interrupt**: INPUT, a frame wait, SYNC, TEMPR(), a network or Bluetooth command. These have never taken interrupts; the interrupt runs when the wait ends. The other threads' interrupts are not held up;
- **another thread is between THREAD HOLD and THREAD RELEASE**;
- **another interrupt routine is running**: routines do not interrupt each other.

Ticks are counted, keys are kept in the keyboard's buffer and pin changes are remembered, so nothing is lost while an interrupt waits. (A SETTICK that falls more than one period behind takes the missed ticks as one, as it always has.)

When its thread ends

The interrupts a thread set up stay set up after it ends, and their routine then runs in the main program. A thread that should take its interrupts with it turns them off before it ends (SETTICK 0, 0, n and so on).

If two threads set up the same SUB as an interrupt routine, it runs in the one that set it up last.

Inside an interrupt routine

No other thread runs while an interrupt routine runs, so a routine cannot wait for something another thread must do. These are errors in a routine: THREAD YIELD, THREAD LOCK of a lock another holds, THREAD

WAIT for something not yet so, and a WEB or BLUETOOTH command while another thread is using the network ("is in use by thread n").

The Network and Bluetooth

These apply only to a Colour Maximite 2 with the WiFi module.

The network belongs to one thread

The network's commands (WEB OPEN TCP CLIENT, TCP CLIENT REQUEST, WEB NTP, WEB MQTT, WEB PING, WEB TRANSMIT and the rest) share one set of state inside the firmware: one TCP client connection, one MQTT client, one timer for their waits. So the network belongs to one thread at a time:

- The main program may use the network first, to set up: WEB CONNECT, WEB NTP and the like at the start of the program do not tie it to the main program.
- The first thread other than the main program that uses a WEB command then has the network, until that thread ends.
- A WEB command in any other thread, the main program included, is then the error "The network belongs to thread n".
- When the owner ends, the network is free again, and the next thread to use it has it.

The interrupt routines the owning thread set up (WEB TCP INTERRUPT, MQTT's and UDP's) run in that thread, so they may use the network too.

If a thread's first WEB command comes while the main program's setup command is still under way, the thread waits for it to finish.

Waiting on the network

A WEB command that waits for an answer lets the other threads run meanwhile. A request with a five second timeout to a server that does not answer costs the rest of the program nothing:

```
Sub Fetcher
  Do
    On Error Skip
    WEB OPEN TCP CLIENT "example.com", 80, 5000
    If MM.Errno = 0 Then
      WEB TCP CLIENT REQUEST req$, buff%(), 5000
      WEB CLOSE TCP CLIENT
      Flag(5) = 1                   ' new data
    EndIf
    Pause 60000
  Loop
End Sub
```

The thread that waits for the network takes no interrupt while it waits, as a WEB command never did; the other threads' interrupts carry on.

Bluetooth

Bluetooth is the same. Its scan, its serial connection (COM8) and its keyboard each keep one state, so BLUETOOTH commands belong to one thread, by the same rule as the network: the main program may use them to set up, the first other thread to use one has Bluetooth until it ends, and a BLUETOOTH command in any other thread is the error "Bluetooth belongs to thread n". BLUETOOTH CONNECT, SCAN and the other commands that wait let the other threads run.

Using COM8 once it is open (PRINT #, INPUT #) is like using any COM port: it is shared, and best given to one thread.

Errors and Debugging

An error ends the program

An error in any thread that is not caught by ON ERROR ends the whole program, as an error always has. The message says which thread it was in:

```
Error in thread 3, line 40: Index out of bounds
```

ON ERROR is each thread's own: ON ERROR SKIP in one thread skips only that thread's next error, and MM.ERRNO and MM.ERRMSG$ are each thread's.

The list of threads

After an error or Ctrl-C, the firmware lists each thread with the line it was at and what it was doing:

```
Error in thread 3, line 40: Index out of bounds

Main program: line 18, in PAUSE
Thread 1: line 7, waiting for lock 2 (thread 3 has it)
Thread 3: line 40, running
```

This shows where every part of the program was, and, for a program that has stopped making progress, which thread waits for what. Ctrl-C on a program that seems stuck is the quickest way to find a deadlock.

CONTINUE is not possible after a program with threads has stopped: their stacks have gone.

THREAD LIST

THREAD LIST prints the same list while the program runs, with each thread's stack: the most it has used, and its size. Both lists show a thread's priority when it is not 1, and "priority 1 (7 while its lock is waited for)" for a thread that has a longer turn for the moment because a thread at priority 7 waits for a lock it holds.

```
Main program: line 22, running, stack used at most 3456 of 27720
Thread 3: line 8, in PAUSE, stack used at most 660 of 16384
```

This is the way to choose a thread's SIZE: run the program through its deepest work, then look at THREAD LIST (see *Memory and Limits*).

TRACE

TRACE ON prints each line's number as it runs, in square brackets. A line run by a thread other than the main program is shown with the thread's number:

```
[23][#1:13][#1:14][24]
```

Here the main program ran line 23, thread 1 ran lines 13 and 14, and the main program went on at line 24. TRACE LIST shows the last lines run, by every thread, in the order they ran.

MEMORY

In a program with threads running, MEMORY has a last line with the memory of their stacks and records:

```
  48K      1 Thread (its stack and the records)
```

OPTION PROFILING

With OPTION PROFILING ON, LIST PROFILE gives each line's count and average time. With threads, a line's time is only the time its own thread spent on it: the time other threads ran while it waited is theirs. A main program in PAUSE 300 while a thread counts shows a few microseconds for the PAUSE line, and the thread's lines show where the 300 ms went.

Stack overflow

A thread's stack holds its SUB and FUNCTION calls. A FUNCTION that calls itself too deeply, or a thread given too small a SIZE, can run out:

```
Error in thread 2, line 12: Stack overflow
```

The cure is a bigger SIZE for that thread, or less recursion. THREAD LIST shows how close each thread has come.

Common mistakes

- **A thread that never ends and a main program that does.** The main program reaching its end ends the program, and every thread with it. Put a loop, or THREAD WAIT, at the end of the main program.
- **Two threads reading the keyboard.** Each gets some of the keys. Let one thread read it.
- **Two statements on shared data without a lock.** See *Sharing Data Safely*.
- **A global variable as a thread's loop counter.** FOR i = 1 TO 10 in a SUB that two threads run, with a global i, is one counter for both. Make it LOCAL, as in an interrupt routine; OPTION EXPLICIT finds the ones that were never declared.
- **A lock left taken.** EXIT SUB between THREAD LOCK and THREAD UNLOCK: the next thread to want the lock waits for ever. Ctrl-C shows which thread has it.
- **A test of a value another thread is changing**, followed by an action on it: the value may have changed in between. Use THREAD HOLD or a lock around both.
- **A busy loop waiting for another thread** (`Do While done = 0 : Loop`). It works, but uses the processor the other thread needs. THREAD WAIT n, or THREAD WAIT FLAG, waits without.
- **Too little stack.** A thread of the default 20 KB can call about nine FUNCTIONs inside each other.

Memory and Limits

What a thread takes

A thread takes two blocks of memory while it runs, both freed when it ends:

- **its stack**: 20 KB unless SIZE or THREAD SIZE says otherwise. It comes from the fast memory inside the processor while plenty of that is free, and from the SDRAM otherwise;
- **its record**: the thread's own lists of its FOR loops, DO loops, calls, temporary strings and local variables, with its other state. Its size follows LOCALS and NEST: about 6 KB by default, about 2 KB at the smallest.

Thread	Stack	In all
The default (SIZE 20 LOCALS 48 NEST 20)	20 KB	about 26 KB
SIZE 12 LOCALS 8	12 KB	about 17 KB
SIZE 12 LOCALS 8 NEST 4	12 KB	about 14 KB

On a Generation 1, with about 4 MB of memory for programs, and on a Generation 2 or 3, with about 15 MB, sixteen threads fit easily. A thread that runs EXECUTE takes another 512 bytes from its first EXECUTE to its end.

The stack

A thread's stack holds the SUBs and FUNCTIONs it is inside. SUB calls take almost none of it. FUNCTION calls take about 1.4 KB each (less with OPTION COMPILE ON, more for a string FUNCTION). The commands themselves need room below that, up to about 6 KB for the largest (LOAD PNG), which every stack keeps free.

With the default 20 KB, a thread can be about nine FUNCTIONs deep. To choose a size:

1. Run the program through the deepest work each thread does. 2. Look at THREAD LIST: "stack used at most n of m". 3. Give each thread SIZE enough for its most, with a few KB to spare.

A thread that only loops and pauses, and calls SUBs, needs very little: SIZE 12, the smallest, is plenty.

Local variables

Each thread has room for a number of LOCAL variables at once: 48 unless LOCALS or THREAD SIZE says otherwise. Every LOCAL, every parameter of every SUB and FUNCTION it is inside, and a FUNCTION's own result each take one. A FUNCTION that calls itself 14 deep with one parameter and one LOCAL needs 3 x 15 = 45. Running out is the error "Not enough Local variable memory".

All the threads' local variables come from one table of 512, which the main program shares, so the more each thread has, the fewer threads can run: with 48 each, about ten. LOCALS 8 for small threads lets all sixteen run.

Loops and calls (NEST)

Each thread keeps its own lists of the loops and calls it is inside, and NEST says how long they are: 20 unless NEST or THREAD SIZE says otherwise. With NEST d a thread may be:

- d FOR loops inside each other;
- d DO loops inside each other;
- d SUB, FUNCTION and GOSUB calls deep, counted from its own SUB. An interrupt routine that runs in the thread counts as one of them.

Going deeper is the usual error: "Too many nested FOR loops", "Too many nested DO or WHILE loops", "Too many nested SUB/FUN" or "Too many nested GOSUB". NEST also sets how many temporary strings the thread's expressions may hold at once (64 and 16 more for each level), which no ordinary statement comes near; running out is "Not enough temporary memory".

The main program's lists are the ones it always had, 100 deep, whatever the threads are given. A thread that loops, pauses and calls a SUB or two needs no more than NEST 4, the smallest.

Limits

What	Limit

Threads at once	16 (numbered 1 to 16), and the main program
Arguments to THREAD START	16
Stack (SIZE)	12 to 512 KB, 20 by default
Local variables (LOCALS)	8 to 256, 48 by default
Loops and calls deep (NEST)	4 to 100, 20 by default
Time slice (THREAD SLICE)	1 to 100 ms, 1 by default
Priority (PRIORITY, THREAD PRIORITY)	1 to 10, 1 by default
Locks	16
Flags	64 (FLAG(0) to FLAG(63))

What cannot be done in a thread

These are refused in a thread other than the main program ("Not allowed in a thread"):

- RUN, CHAIN, FLASH RUN, FLASH CHAIN, RAM RUN, RAM CHAIN and LIBRARY LOAD, which replace the program;
- LIST, DEBUG BREAK, XMODEM and ZMODEM, which take over the console;
- MsgBox() and GUI CLICK, which take interrupts while they wait in a way only the main program can.

These are refused while any thread is running ("Not allowed while a thread is running"), as another thread may be using the variable or memory they change:

- ERASE, REDIM and CLEAR;
- SAVE CONTEXT and LOAD CONTEXT.

And these are not possible:

- THREAD START at the command prompt;
- CONTINUE after a program with threads has stopped.

Things to know

- **FRAMEBUFFER CLOSE** is refused while any thread writes to the framebuffer ("Framebuffer is set for write").
- **MODE** gives every thread the new mode's print position, colours, font and write page.
- **CLS** puts only the running thread's print position back to the top.
- **TIMER = 0** in one thread sets the timer for all.
- **OPTION settings** (BASE, DEFAULT, EXPLICIT, ANGLE and the rest) are the program's: set them before the threads start.
- **Loadable programs** (RUN "name.app"), the C compiler's programs and MicroPython do not use threads; a CSUB runs whole, with no switch inside it.

Worked Programs

Balls and a clock while the user types

Four balls bounce, each in a thread running the same SUB with its own arguments; a clock in the corner is a fifth thread; the main program asks for lines of text. Everything goes on while the user types.

```
Option Explicit
Dim Integer i, go = 1
Dim a$
Dim Integer col(4) = (0, RGB(red), RGB(green), RGB(yellow), RGB(cyan))

Sub Ball(x As Float, y As Float, dx As Float, dy As Float, c As Integer)
  Do While go
    Circle x, y, 5, 1, 1, 0, 0                    ' rub the old one out
    x = x + dx : y = y + dy
    If x < 10 Or x > MM.HRes - 10 Then dx = -dx
    If y < 130 Or y > MM.VRes - 10 Then dy = -dy
    Circle x, y, 5, 1, 1, c, c
    Pause 20
  Loop
End Sub

Sub Clock
  Do While go
    Text MM.HRes - 1, 0, Time$, "RT", 1, 1, RGB(white), RGB(blue)
    Pause 250
  Loop
End Sub

CLS
For i = 1 To 4
  Thread Start i Size 12 Locals 8, Ball, 100 * i, 150 + 60 * i, 2 + i, 1 + i, col(i)
Next
Thread Start 5 Size 12 Locals 8, Clock
Do
  Print @(0, 40) Space$(70);
  Print @(0, 40);
  Input "Type something (Q to quit)"; a$
  Print @(0, 70) "You typed: "; a$; Space$(60)
Loop Until UCase$(a$) = "Q"
go = 0
End
```

Each ball has its own position and speed as the parameters of its SUB, so one SUB serves all four. Each CIRCLE and TEXT carries its own colour, so the threads cannot disturb each other's colours, and each thread's print position is its own, so the clock does not move the line being typed.

A long job with a live screen

The picture is drawn by a thread. The main program reads the keys: P pauses the drawing, R lets it go on, Q stops it.

```
Option Explicit
Dim Integer done, row
Dim k$

Sub Picture
  Local Integer x, y
  For y = 0 To MM.VRes - 1
    For x = 0 To MM.HRes - 1
      Pixel x, y, RGB(x And 255, y And 255, (x Xor y) And 255)
    Next
    row = y
  Next
  done = 1
End Sub

CLS
Thread Start 1, Picture
Do While done = 0
```

```
  k$ = UCase$(Inkey$)
  If k$ = "P" Then Thread Pause 1
  If k$ = "R" Then Thread Resume 1
  If k$ = "Q" Then Exit Do
  Pause 50
Loop
Thread Stop 1                 ' (does nothing if the thread has ended)
Print @(0, 0) "Ended at row"; row
```

Workers sharing a total

Four threads add up parts of an array into one total. The read and write of the total are two statements, so a lock keeps them together.

```
Option Explicit
Dim Integer i, total
Dim Integer d(3999)
For i = 0 To 3999 : d(i) = i : Next

Sub Adder(first%, last%)
  Local Integer k, part
  For k = first% To last%
    part = part + d(k)
  Next
  Thread Lock 1
  total = total + part        ' (one statement: the lock is for the
  Thread Unlock 1             '  pattern; a longer job would need it)
End Sub

For i = 0 To 3
  Thread Start i + 1, Adder, i * 1000, i * 1000 + 999
Next
For i = 1 To 4 : Thread Wait i : Next
Print "Total"; total          ' 7998000
```

A sensor logger

One thread samples a pin every 100 ms with its own SETTICK; another writes the samples to the SD card every ten seconds; the main program shows the latest reading.

```
Option Explicit
Dim Integer n, latest
Dim Float buf(99)

Sub Sampler
  SetTick 100, Sample, 1
  Do : Pause 1000 : Loop          ' its interrupt runs in this thread
End Sub

Sub Sample
  latest = Pin(7)
  Thread Hold                     ' the index and the store together
  buf(n Mod 100) = latest
  Inc n
  Thread Release
End Sub

Sub Writer
  Local Integer k, done
  Open "log.csv" For Append As #2
  Do
    Pause 10000
    Do While done < n
      Print #2, buf(done Mod 100)
      Inc done
    Loop
  Loop
End Sub

SetPin 7, AIN
Thread Start 1 Size 12 Locals 8, Sampler
Thread Start 2 Size 12 Locals 8, Writer
Do
  Text 0, 0, "Latest:" + Str$(latest) + "   "
```

```
  Pause 200
Loop Until Inkey$ <> ""
End
```

The file belongs to the Writer thread alone; the Sampler only fills the array. THREAD HOLD in the routine is not strictly needed (an interrupt routine runs whole, with no other thread), but it shows the pattern for code that is not a routine.

A web request while a game runs

The network belongs to one thread, which fetches a value every minute and leaves it in a global. The rest of the program never waits for the network.

```
Option Explicit
Dim temperature$ = "?"
Dim Integer buff%(512)

Sub Weather
  Local req$ = "GET /temp.txt HTTP/1.1" + Chr$(13) + Chr$(10) + "Host: 192.168.1.10" +
Chr$(13) + Chr$(10) + Chr$(13) + Chr$(10)
  Do
    On Error Skip 3
    WEB OPEN TCP CLIENT "192.168.1.10", 80, 3000
    WEB TCP CLIENT REQUEST req$, buff%(), 3000
    WEB CLOSE TCP CLIENT
    If MM.Errno = 0 Then temperature$ = LGetStr$(buff%(), LLen(buff%()) - 4, 5)
    Pause 60000
  Loop
End Sub

Thread Start 1, Weather
Do
  ' ... the game ...
  Text 0, 0, "Outside: " + temperature$
  Pause 20
Loop
```

The address, the request and the way the reply is read depend on your server; the shape of the program is the point.

Appendix A – The Commands in Brief

Command or function	What it does
THREAD START n [SIZE kb] [LOCALS m] [NEST d] [PRIORITY p], sub [, argument ...]	Starts thread n (1 to 16) running the SUB, with up to 16 arguments by value (a global array by name). SIZE 12 to 512 KB, LOCALS 8 to 256, NEST 4 to 100, PRIORITY 1 to 10, for this thread only
THREAD EXIT	Ends the thread that runs it
THREAD STOP n	Ends thread n where it stands (nothing if it is not running)
THREAD YIELD	Gives up the rest of the time slice
THREAD PAUSE n	Holds thread n, and its interrupts
THREAD RESUME n	Lets thread n go on
THREAD SLICE ms	The time slice, 1 to 100 ms (1 at RUN)
THREAD PRIORITY n, p	The priority of running thread n (0 the main program): the time slices in its turn, 1 to 10 (1 at RUN). A lock's holder has the turn of a thread that waits for it
THREAD SIZE kb [, locals [, nest]]	The stack, local variables and depth of loops and calls of threads started without SIZE, LOCALS and NEST (20, 48 and 20 at RUN)
THREAD LOCK k [, timeout]	Takes lock k (1 to 16), a mutex, waiting for it if another thread has it; "Lock k timed out" after 'timeout' ms (0 tries it without waiting)
THREAD UNLOCK k	Gives lock k up
THREAD HOLD	No other thread runs, and their interrupts wait, until THREAD RELEASE
THREAD RELEASE	Ends THREAD HOLD (they nest)
THREAD WAIT n [, timeout]	Waits until thread n has ended, or for 'timeout' ms
THREAD WAIT FLAG f [, timeout]	Waits until FLAG(f) is 1 (0 to 63), or for 'timeout' ms
THREAD LIST	Each thread's line, state and stack
MM.INFO(THREAD)	The running thread's number (0 the main program)
MM.INFO(THREAD n)	0 not running, 1 running or ready, 2 in PAUSE, 3 paused, 4 waiting
MM.INFO(THREAD LOCK k)	The thread holding lock k (0 the main program), or -1

The error messages

Message	Meaning
Thread n is running	THREAD START of a number already in use
Thread n is not running	THREAD PAUSE or RESUME of a thread that is not running
x is not a SUB	THREAD START names something that is not a SUB
Invalid at the command prompt	THREAD START typed at the prompt
Only a global array can be passed to a thread	A SUB's own array given to THREAD START
Too many arguments for a thread	More than 16 arguments to THREAD START
Arguments too long for a thread	The new thread's call does not fit a line (many arrays with long names)
Not enough memory for a thread	No room for the thread's stack and record
Not enough local variables for a thread	The local variable table is full (smaller LOCALS)
Not in a thread	THREAD EXIT in the main program

Not allowed in a thread	RUN, CHAIN, LIST and the others of *Memory and Limits*
Not allowed while a thread is running	ERASE, REDIM, CLEAR, SAVE or LOAD CONTEXT
Lock k timed out	THREAD LOCK k, timeout: the lock was not free in time
Lock k is already held by this thread	THREAD LOCK of a lock the thread holds
Lock k is not held by this thread	THREAD UNLOCK of a lock the thread does not hold
The threads are not held	THREAD RELEASE with no THREAD HOLD
Cannot wait while the threads are held	A wait inside THREAD HOLD
Cannot wait in an interrupt routine	THREAD LOCK or WAIT that cannot end at once, in a routine
A thread cannot wait for itself	THREAD WAIT of the running thread's number
The network belongs to thread n	A WEB command in a thread that does not own the network
Bluetooth belongs to thread n	A BLUETOOTH command in a thread that does not own Bluetooth
The network (or Bluetooth) is in use by thread n	A WEB or BLUETOOTH command in an interrupt routine while another thread's is under way
Invalid in an interrupt routine	THREAD YIELD in a routine
Too many interrupt routines set up in threads	More than 64 different routines set up by threads
Thread n overflowed its stack	Thread n's stack was overrun (larger SIZE)
Stack overflow	A thread's FUNCTION calls went too deep for its stack
Not enough Local variable memory	A thread made more LOCALs than its LOCALS
Too many nested FOR loops (or DO or WHILE loops, SUB/FUN, GOSUB)	A thread's loops or calls went deeper than its NEST