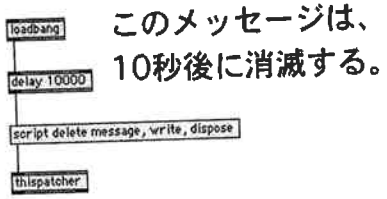
■3-14-18 メッセージ・ボックスを自動消滅させるパッチ




このパッチでは、メッセージ・ボックスにmessageという名前を付けている。そして、loadbangとdelayオブジェクトによってパッチを開いてから10秒後にscript delete messageをthispatcherオブジェクトに送って、メッセージ・ボックスを削除する。さらに、writeをthispatcherオブジェクトに送って、パッチをファイルとして保存する。最後にdisposeによってパッチを閉じる。このようにして、メッセージ・ボックスを削除したあと、メッセージ・ボックスが存在しなくなったパッチをファイルに保存しているわけだ。





## 3-15 デバッグのテクニック

Maxのプログラミングでは、パッチを作成して実行結果を検討し、動作不良があればパッチを改良して再度実行する、といった一連の作業を途切れなく行うことができる。このために、実際には明確に意識しないままデバッグ作業を行っていることが多いが、しかしながら、どのようにデバッグを行うか、その典型的な手法を理解しておくに越したことはない。ここでは、デバッグの段階的な手順や、簡単なメッセージの確認方法、そしてMaxに内蔵されたデバッガーであるトレース機能について説明する。

### 周辺環境の確認

一般に、コンピューター・プログラムの動作不良やその原因をバグと呼び、バグを取り除いて正しく動作するように改良する作業をデバッグと呼ぶ。また、デバッグ作業を支援するツールはデバッガーと呼ばれる。

パッチが正しく動作しない場合に、さまざまな原因が考えられる。ほとんどの場合はパッチ自体に原因があるが、パッチ以外の原因も検証するのがよいだろう。例えば、音が鳴らないとか、表示がされないといった、根本的な障害があるのかもしれない。このような場合には、似たような処理を行うチュートリアルやサンプルのパッチを動作させてみるとよい。あるいは、ヘルプのパッチでも事足りるかもしれない。例えば、MIDI音源から音が聞こえないなら、MIDI音源を鳴らすチュートリアルのパッチを試してみることだ。チュートリアルなどのパッチは、本来正しく動作するはずなので、これが正しく動作しないとすれば、周辺機器やコンピューターの設定に間違いがある可能性が高い。機器の接続状態や各種の設定をチェックしてみよう。

また、頻度は少ないが、Maxアプリケーションやコンピューターのシステムが動作不良になっていることも考えられる。そういったときは、作成中のパッチを閉じたあとに再度開けば、正しく動作する可能性がある。Max自体やコンピューターを再起動した上で、再度パッチを試してみるのもよいだろう。システムフォルダの初期設定フォルダの中にあるMax 4 Preferencesという初期設定ファイルをゴミ箱に捨てた上で、Maxを起動しなおすことで、Maxの動作が正しくなることもある。

なお、正常に動作しているパッチであっても、パッチを開き直して動作を確かめることは有益だ。問題なく動作しているパッチを開き直すと、正しく動作しない場合もあり得る。これは必要な設定が行われていないことが原因かもしれない。そのような場合には、loadbangオブジェクトを利用して自動的に設定を行うような工夫をすべきだろう。

また、自分のコンピューター環境では正しく動作しても、他の環境では正しく動作しない可能性も考えておかなければならない。そのため、作成したパッチを配布する場合は、異なる環境でも正しく動作することを確かめておくべきである。

## オブジェクトの確認

Maxアプリケーションやコンピューターと周辺機器に問題がないとすれば、自分が作成しているパッチにバグが含まれていることになる。パッチの動作不良の原因を調べるには、まず、パッチがどのような処理を行っているか、それぞれのオブジェクトがどのような働きをしているかを、よく考えてみるのがよい。Maxには膨大な数のオブジェクトがあり、それぞれがさまざまなメッセージに従って異なる動作をする。似たような動作をするオブジェクトやメッセージもがあるが、それらを正しく用いなければならない。

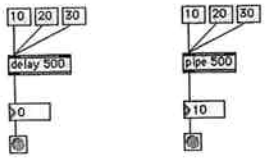
例えば、数値を時間的に遅らせる処理をしたい場合に、delayオブジェクトを使ったとしよう。そして、どのような数値をdelayに送っても、0が出力されているように見える。これは、delayオブジェクトはbangメッセージを遅らせるオブジェクトであり、数値を遅らせることはできないからだ。しかも、delayオブジェクトは数値を受け取ると、それを遅延時間の設定として扱い、その時間後にbangメッセージを出力する。したがって、パッチやオブジェクトとしては正しく動作しているので、エラー・メッセージも出ない。しかし、数値を遅らせて出力するという目的は実現できない。このような場合は、オブジェクト自体の機能を確かめない限り、どのようなデバッグ作業を行っても解決しない。具体的には、各オブジェクトのヘルプやリファレンス・マニュアルを参照して、オブジェクトの使用方法を検討しなければならないだろう。上記の場合は、delayではなくpipeオブジェクトを用いるのが正しい。

複数のオブジェクトを組み合わせた処理が正しく動作しない場合は、個々のオブジェクトの機能だけでなく、パッチ全体としての動作を検討しなければならない。特に間違いが起きやすいのは、メッセージの入出力順序やオブジェクトの処理順序である。





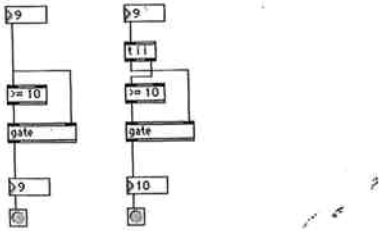
■3-15-1 数値を遅らせる処理の誤った例(左)と正しい例(右)




ほとんどのオブジェクトはright-to-left orderに従うので、そのルールに基づいて処理がどのように行われるか検討してみよう。ただし、一部right-to-left orderに合致しない動作をするオブジェクトもあり、注意が必要である。3-15-2のパッチでは、10以上の数値を通過させる処理を行おうとしている。しかし、左側のパッチでは、上部のナンバー・ボックスを動かして、0から次第に数値を上げていくと、0から10までの数値は通過せず、11以上の数値が通過する。逆に、大きな数値から小さな数値へと下げていった場合は、9までの数値が通過し、8以下の数値が通過しない。

このような動作不良の原因は、gateオブジェクトの第2インレットに数値が送られたあとに、数値比較の結果がgateの第1インレットに送られるからである。つまり、数値を通過させるか否かを判断する前に、その数値がgateオブジェクトを通過しようとするからだ。それ自体はright-to-left orderに従った動作であるが、処理の目的とgateの動作を考えれば、先に条件判断を行わなければならない。したがって、正しくはtriggerオブジェクトを用いて、条件判断の結果を先にgateオブジェクトの第1インレットに送ったあと、gateオブジェクトの第2インレットに数値を送る必要がある。そのような処理を行っているのが右側のパッチで、このパッチでは正しく10以上の数値だけが通過する。

■3-15-2 10以上の数値を通過させる処理の誤った例(左)と正しい例(右)



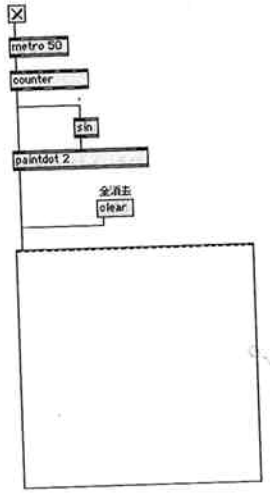
## メッセージの確認

簡単なパッチならオブジェクトやパッチ・コードを目で追いかけて、その動作の状況を推測することができる。だが、複雑なパッチになれば、頭が混乱するかもしれないし、そもそも推測が正しいとは限らない。そこで、パッチの要所要所において、どのようなメッセージが流れているかを表示すれば、処理結果を検討することができるだろう。

具体的には、あるアウトレットからメッセージが出力されたことを確認するには、buttonオブジェクトをつないで、その点滅を確かめればよい。また、数値が出力されるアウトレットには、ナンバー・ボックスやフロート・ナンバー・ボックスをつなげばよい。リストの場合は、unpackオブジェクトで要素ごとに分けて、それぞれをナンバー・ボックスなどで表示させる。シンボルや複数の要素から成るメッセージは、prepend setオブジェクトを通じてメッセージ・ボックスに表示させるといった方法が考えられる。

printオブジェクトを用いれば、どのようなメッセージでもMaxウィンドウに表示させることができるので、広く活用できるだろう。

■3-15-3 サイン波を描く処理の誤った例

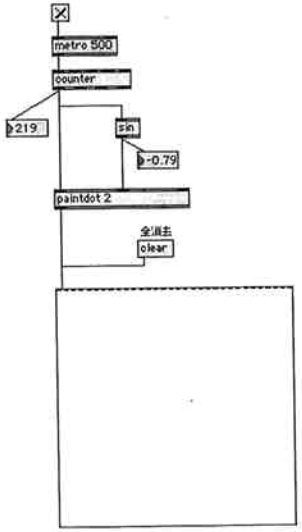







3-15-3の例ではlcdオブジェクトにサイン波を描こうとしている。Maxにはsinというサイン関数のオブジェクトがあり、一般にサイン波は、y=sin(x)という式で描くことができる。xは水平方向のx座標の値、yは垂直方向のy座標の値だ。このことを単純に用いれば、3-15-3のようなパッチになるだろう。しかし実際にはこのパッチのtoggleオブジェクトにチェックを付けてもlcdオブジェクトの上端付近に点線が描かれるだけで、サイン波は描かれない。なぜだろう? そこで、counterオブジェクトからの出力をナンバー・ボックスにつなぎ、sinオブジェクトからの出力をフロート・ナンバー・ボックスにつなぐ。そしてパッチを動作させて、どのような値が出力されるか確かめてみよう。数値の変化が速すぎて確認しにくいなら、metroオブジェクトの動作間隔を長くすればよい。あるいは、printオブジェクトを使ってMaxウィンドウに表示し、パッチの動作を止めてから、どのような数値が出力されたかを調べてもよいだろう。

■3-15-4 オブジェクトからのメッセージを表示



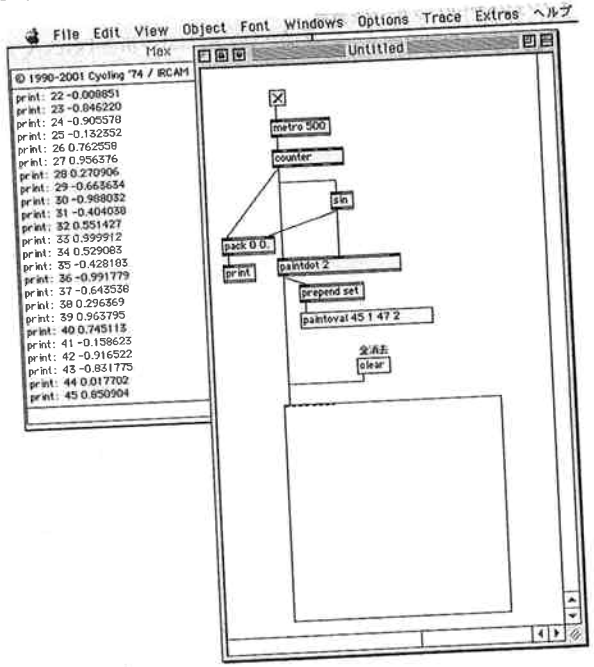

このようにして、オブジェクトから出力される数値を検討すれば、2つの問題点が浮かび上がる。まずx座標の値として用いるcounterの値が、ここでのlcdオブジェクトの

横幅である200を超えてしまうことだ。200以上のx座標であれば、lcdオブジェクトの領域外に描くことになり、実際には何も描画されない。

もう1つの問題はsinオブジェクトの出力は－1.0から1.0までの範囲で変化することだ。このような数値を用いて描画しても、1ピクセル以下の変化は反映されず、y座標の値として0以下はlcdオブジェクトの領域外なので描画されない。このような理由で、lcdオブジェクトの上部に点線が描かれたわけだ。

ちなみに、sinオブジェクトからの出力が実数だが、ナンバー・ボックスで数値を確認すれば、常に0が表示されることになる。これは表示の方法自体が間違っているわけで、これではバグの原因解明が困難になってしまう。このように、どのようにデバッグを行うかという方法にも細心の注意を払う必要がある。

■3-15-5　異なる方法でのメッセージの表示

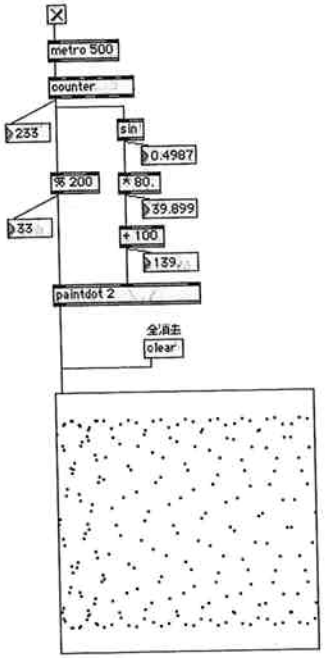







3-15-5の例はメッセージを異なる方法で表示させるものだ。ここでは、counterオブジェクトの出力とsinの出力をpackでまとめ、printオブジェクトでMaxウィンドウに表示させている。このようにすれば、x座標とy座標の値の関係が分かりやすくなり、それぞれの値の変化をじっくりと調べることができる。また、paintdotオブジェクトからのメッセージは、prepend setオブジェクトでメッセージ・ボックスに表示している。paintdotオブジェクトが受け取った座標値によって、lcdオブジェクトへのメッセージがどのように出力されているかを把握することができる。

■3-15-6　x座標とy座標の値を調整した処理




さて、これまでに見つかった問題点の解決策は次のようになる。

まず、x座標の値がlcdオブジェクトの横幅を超えてしまう点は、counterのアーギュメ

ントを指定してcounter 0 0 199とすればよい。これでx座標の値は0から199までを繰り返し、lcdオブジェクトの領域内で変化することになる。あるいはcounterからの値を% 200オブジェクトを使って、200で割った余りをx座標の値として用いる方法も考えられる。ここでは% 200を用いることにする。

また、sinオブジェクトの出力が−1から1までであり、ほとんど変化が現れない問題については、lcdオブジェクトの縦幅が200であるので、sinオブジェクトからの出力を80倍して100を足せばよい。−1から1までの変化を80倍すると、−80から80までの範囲になり、100を足せば20から180までの範囲で変化することになる。これでy座標の値が、lcdオブジェクトの真ん中を中心に両端近くまで変化する。ただし、80倍は * 80.0として実数計算を行わなければならない。先にも説明したが、整数と実数の違いが処理結果に大きな影響を及ぼすことがあり、ここでの処理もその一例だ。一方、+ 100は整数計算で構わない。

さて、こうしてバグをなくした3-15-6のパッチでも、まだ描画はサイン波らしくない。それぞれのオブジェクトからの出力をナンバー・ボックスなどに表示している数値を確認すれば、y座標の値の変化が大きいために、連続的に見えないことが分かるだろう。

これは、sinオブジェクトへ入力する数値がラディアン単位であることを思い出せば解決できる。ラディアンは全周である360°を2πとする単位なので、サイン波の1波長は0から2π (6.28318)に当たる。つまり、x座標の約6ピクセルごとに1波長ずつ描いているわけだ。したがって、サイン波らしい表示にするにはsinオブジェクトに入力する数値が増加する割合を少なくすればよい。

そのために、counterオブジェクトの出力を20で割ってから、sinオブジェクトに入力することにしよう。もちろん/ 20.として実数計算する必要がある。ちなみに、20という数値は適当に決めたものだが、割る数値が大きくなるほど波長も長くなる。

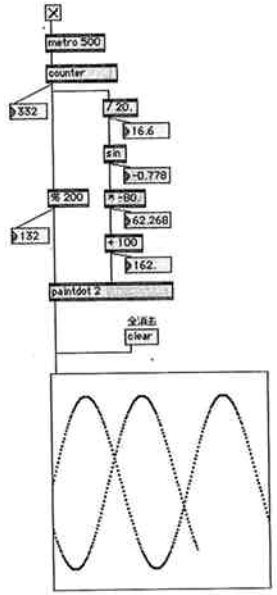
これで、ようやくサイン波らしい描画になった(3-15-7)。さらにこのパッチでは、数学的な座標系とlcdオブジェクトの描画する座標系は縦方向が逆なので、sinオブジェクトが出力する値に80.ではなく−80.を掛けている。これで、数学的な座標系とも一致する。

なお、パッチが正しく動作するようになれば、メッセージを確認するためのナンバー・ボックスなどは削除しても構わない。処理の負荷が少ないパッチであれば、そのまま残しておいてもよいだろう。また、将来の改良や確認に備えて、オブジェクトは残しておき、パッチ・コードだけ切断しておく方法も考えられる。





■3-15-7 サイン波らしい描画となった処理

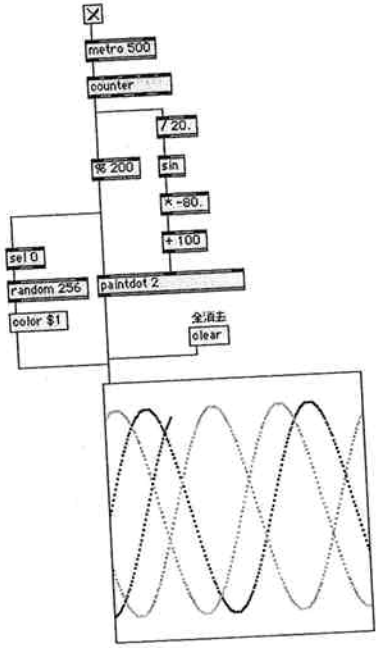



## ❷ 処理順序の確認

このようにしてサイン波が描けるようになったので、これを発展させてlcdオブジェクトの左端に戻ったときに色を変えて描くように改良するとしよう。このためには、描画位置がlcdオブジェクトの左端に戻るのは、x座標の値が0になるときなので、% 200の出力をsel 0へ送る。sel 0は受け取った数値が0であればbangメッセージを出力するので、これを受けてrandom 256によってランダムな数値を出力する。そしてcolor $1のメッセージ・ボックスを使ってlcdオブジェクトの描画色を設定すればよいだろう。

このようにパッチを改良すれば、1回ごとにランダムに色を変えながら、サイン波が描かれるようになる(3-15-8)。

■3-15-8 色を変えながらサイン波を描く処理




これで一見正しく処理が行われているように見えるが、よく観察すると左端のドットの色が前回の色のままであり、2つ目から色が変化していることが分かる。つまり、描画位置が左端になった時点で色が変更されないというバグが発生したわけだ。

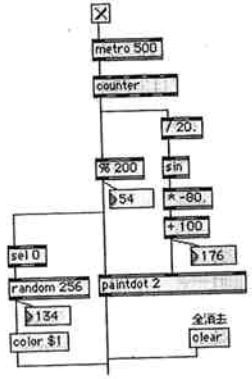
そこで、これまでと同じようにナンバー・ボックスを使ってメッセージの内容を表示し、問題点を検討してみることにしよう。

だが、結論から言って、おそらくメッセージの内容を検討するだけでは原因は解明されないだろう。問題になっているのは、左端で色を変えてドットを描く瞬間であり、これは一瞬のうちに処理される。したがって、ほぼ同時に表示されるナンバー・ボックスの





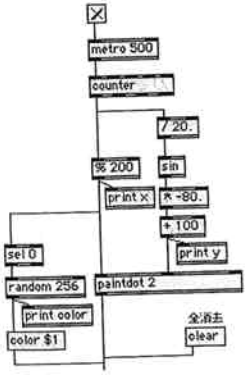
■3-15-9 ナンバー・ボックスによってメッセージの内容を表示




表示を見ているだけでは、どこに原因があるのか判別できないはずである。

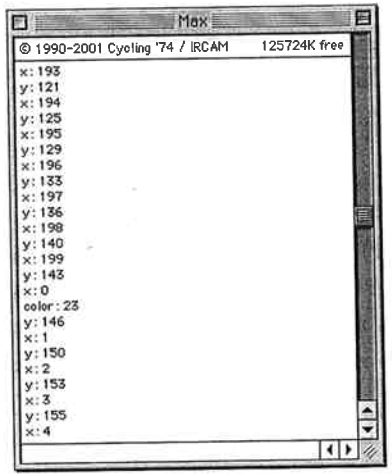
このような場合は、メッセージの内容は正しいが、処理順序やメッセージが送られる順序に問題があると想像できる。そこで、printオブジェクトを用いてメッセージの内容をMaxウィンドウに表示してみよう。ただし、複数のprintオブジェクトを用いる場合は、どのprintオブジェクトが表示したメッセージなのか判断ができないかもしれない。

■3-15-10 printオブジェクトによってメッセージの内容を表示



そんなときは、それぞれのprintオブジェクトに異なるアーギュメントを与えればよい。printオブジェクトはアーギュメントを先に付けてメッセージを表示する。例えば、print xが28というメッセージを受け取った場合には、x:28と表示する。ではprint x、print y、print colorという3つのprintオブジェクトによって、それぞれのメッセージをMaxウィンドウに表示させてみよう。こうして表示させたメッセージは3-15-11のようになる。上部に表示されるメッセージほど、時間的に先に出力されたメッセージだと考えればよい。

■3-15-11　Maxウィンドウに表示されたメッセージ




この表示結果を検討すれば、x:0の次にcolor:23が出力されていることが分かる。つまり、左端でx座標の値0がpaintdotオブジェクトに送られてドットが描かれたあとに、色を変更するメッセージがlcdオブジェクトに送られていることになり、これが左端で色が変更されない原因だと判断できるだろう。そこで、triggerオブジェクトを使ってメッセージを送る順序を設定する。つまり、paintdotオブジェクトにx座標の値を送る前に、x座標の値が0であるか否かを判断し、0であれば色を変えるようにするのである。

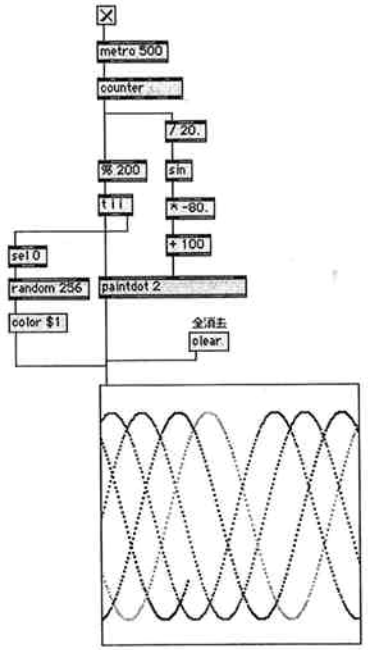
このように改良したパッチが3-15-12だ。これで、左端でも正しく色を変えて描画されるようになる。メッセージがどのような順序で送られ、どのような順序で処理が行われて





いるかを知りたいときは、複数のprintオブジェクトを適切な箇所に用いればよいだろう。それぞれのprintオブジェクトに適切なアーギュメントを指定することも大切だ。

■3-15-12　正しく色を変えながらサイン波を描く処理



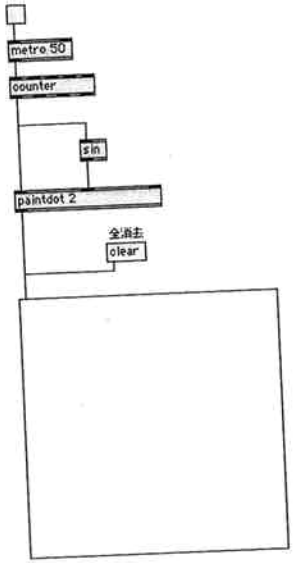

## ❷ トレース機能の利用

これまでに説明したように、ナンバー・ボックスやメッセージ・ボックスを用いれば、どのようなメッセージが流れているかを確認することができる。

また、printオブジェクトによってメッセージをMaxウィンドウに表示することも可能であり、Maxウィンドウに表示されるメッセージの並びによって、処理の順序も把握できる。簡単なデバッグならば、このような方法で事足りるが、より複雑なデバッグ作業を行わな

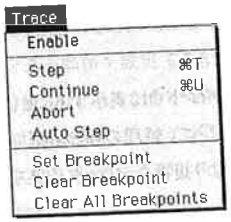
ければならない場合には、Maxに備わったデバッガーであるトレース機能を利用するのがよいだろう。以前のパッチを使って、トレース機能を使い方を見ていこう。

■3-15-13　トレースするパッチ




トレース機能はTraceメニューによって操作を行う。通常、TraceメニューはEnableという項目だけが有効で、それ以外の項目はグレー表示されて選択できない。これはトレースを行っていない状態を意味する。

■3-15-14　通常のTraceメニュー

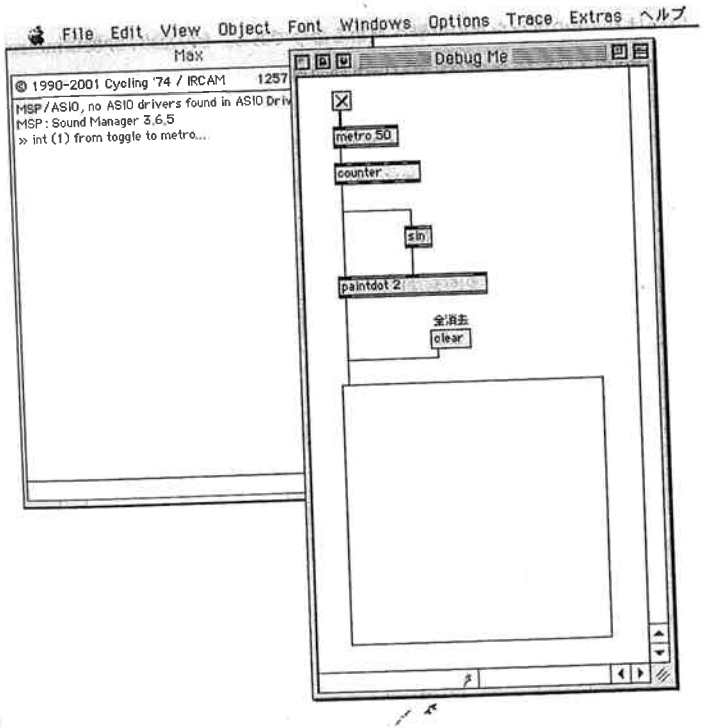






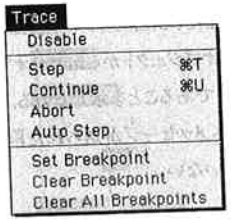

そこで、Enableを選択して、トレース機能を有効にする。これでEnableがDisableという表示に変わり、トレース機能が働いている状態になる。この状態で、toggleオブジェクトをクリックしてみよう。toggleオブジェクトからmetroオブジェクトへのパッチ・コードが点滅する太線となって強調表示される。これは、toggleオブジェクトからmetroオブジェクトへメッセージが流れたことを示している。また、Maxウィンドウにはint（1）from toggle to metro...と表示される。これもtoggleオブジェクトからmetroオブジェクトへメッセージが流れ、それはintメッセージで値が1であることを示している。パッチの実行状態としては、toggleオブジェクトからmetroへメッセージが送られた段階で一時停止しており、metroオブジェクトはまだ動作していない。

■3-15-15　トレース中のパッチとMaxウィンドウの表示



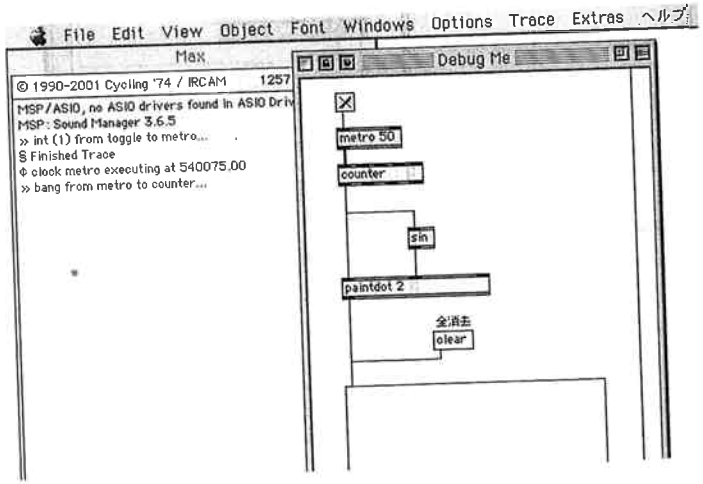
次に、Traceメニューから Stepを選ぶ。Traceメニューは、他の項目も有効になっていることが分かるだろう。

■3-15-16 トレース中のTraceメニュー




Stepはパッチの実行を次の段階へ進める命令である。そのため、今度はmetroからcounterオブジェクトへのパッチ・コードが点滅し、Maxウィンドウにはbang from metro to counter...と表示されるだろう。これらによって、metroからcounterオブジェクトへbangメッセージが送られたことが確認できる。パッチは、metroオブジェクトが動作し、bangメッセージをcounterオブジェクトへ出力した段階で一時停止している。

■3-15-17 ステップを進めたパッチとMaxウィンドウの表示








それより前にFinished Traceと表示されているが、これはtoggleからmetroオブジェクトへメッセージが流れた時点で、一連の処理の流れが終了したことを示している。つまり、toggleから出力されたintメッセージは、metroオブジェクトが受け取った段階で処理が完了する。そして、metroオブジェクトは新たなbangメッセージを出力することになり、それは受け取ったintメッセージとは異なるメッセージの流れとして扱われるわけだ。また、その次にMaxウィンドウに表示されるclock metro executing at 540075.00は、内部クロックが540075.00のときにmetroオブジェクトが動作したことを示している。

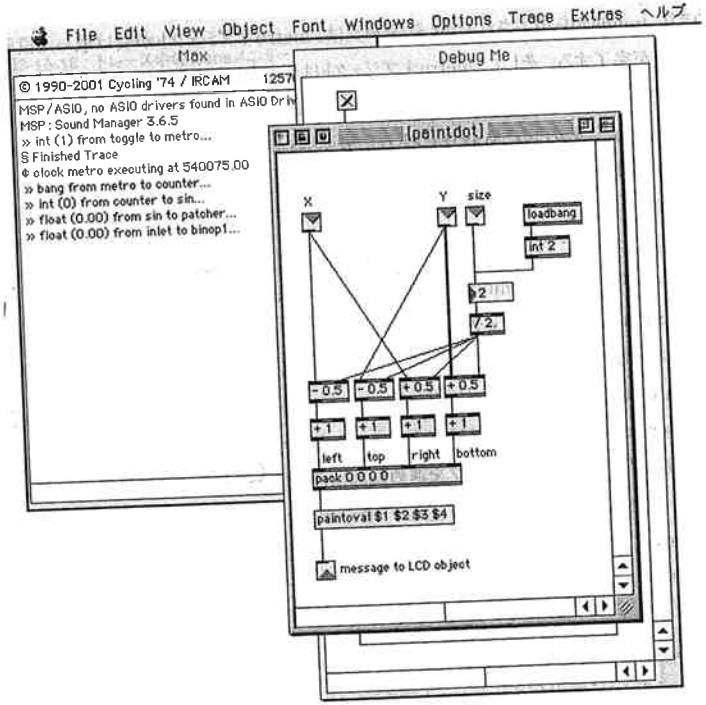
それでは、さらにTraceメニューのStepを選んで、処理を進めていこう。メニューから選ぶのが面倒なら、command-Tのキーボード・ショートカットを用いてもよい。次には、counterからsinオブジェクトへのパッチ・コードが点滅し、int (0) from counter to sin...と表示される。

counterオブジェクトのアウトレットからは2つのパッチ・コードが延びているが、right-to-left orderに従って、sinオブジェクトへのパッチ・コードへ先にメッセージが送られたことが確認できる。さらにステップを進めれば、sinからpaintdotオブジェクトへのパッチ・コードが点滅し、float (0.00) from sin to patcher...と表示される。ちなみに、paintdotはパッチ・オブジェクトなので、paintdotではなくpatcherと表示されている。面白いのは、次のステップではpaintdotオブジェクトのパッチ・ウィンドウが開いて、メッセージが流れたパッチ・コードが点滅することだ。ステップを何段階か進めて、paintdotオブジェクト内のトレースが終われば、元のパッチ・ウィンドウに戻ってトレースが続く。このように階層化されたパッチであっても、自動的に必要なパッチ・ウィンドウを開いてトレース表示が行われるようになっている(3-15-18)。

このようにして、トレース機能によってパッチの実行段階を1ステップずつ進めて、メッセージがどのような順序で流れ、そのメッセージの種類と内容がどのようになっているかを観察することで、バグの解明を行うことができる。

また、TraceメニューのAuto Stepを選べば、パッチの実行が短い時間間隔で自動的に進行していく。パッチ・コードの点滅が次々と進み、そのメッセージの内容は順次Maxウィンドウに表示されていく。ちょうど、スローモーションでパッチの実行が進行し、その様子を観察できるわけだ。

■3-15-18 パッチ・ウィンドウが自動的に開かれ、トレースが行われる




規模の大きなパッチになれば、Stepで1ステップずつ実行段階を進めていくことは面倒になる。問題のある箇所を見つけるまでに、何段階ものステップを進めなければならないからだ。大量のメッセージをチェックし続けるのも困難である。Auto Stepを使っても本質的な解決にはならない。

このような場合には、パッチの中で問題が起こっていると思われる部分を仮定して、ブレイクポイントを設定すればよい。

仮にこのパッチでは、sinオブジェクトの出力に問題があると推測したとしよう。そこで、パッチをアンロック状態にして、sinからpaintdotオブジェクトへ接続されているパッチ・コードをクリックして選択する。そして、TraceメニューからSet Breakpointを選ぶ。これで選択しているパッチ・コードにブレイクポイントが設定された。





次に、TraceメニューのContinueを選べば、パッチは一時停止状態を解除して、通常の実行を始める。Maxウィンドウへのメッセージ内容の表示も行われない。そして、ブレイクポイントを設定したパッチ・コードにメッセージが流れると、パッチは再び一時停止状態になる。この段階のメッセージはMaxウィンドウに表示される。つまり、ブレイクポイントに到るまでの実行が一気に行われ、途中の段階を気にしなくてよいわけだ。ここでは、sinオブジェクトがメッセージを出力する時点で処理が一時停止され、出力されたメッセージがMaxウィンドウに表示される。したがって、Continueを何度か繰り返せば、sinオブジェクトが出力している数値の変化を把握することができる。

なお、ブレイクポイントは複数のパッチ・コードに設定できる。ブレイクポイントの設定を解除するには、パッチ・コードを選択してTraceメニューからClear Breakpointを選べばよい。TraceメニューのClear All Breakpointsを選べば、すべてのブレイクポイントが解除される。

また、一連のメッセージの流れを中断するには、TraceメニューのAbortを選ぶ。トレース機能を止めるには、TraceメニューからDisableを選べばよい。

以上のように、トレース機能を使えば効率的なデバッグ作業ができる。ただし、パッチの実行を一時的に停止することになるので、時間的な処理が重要になるようなデバッグには、トレース機能が向かない場合もあるだろう。そのような場合には、前述したナンバー・ボックスやprintオブジェクトを使ったデバッグを行うことになる。