認知実験における時間制御の問題

仮想デバイス・ドライバを用いたタイマ・ルーチンの検討

下木戸　隆司

2004 年 5 月

JCSS-TR-50

名古屋大学大学院　教育発達科学研究科　心理学系教室

〒 464-8601　名古屋市千種区不老町

m47031a@cc.nagoya-u.ac.jp

日本認知科学会　事務局

〒 464-8601　名古屋市千種区不老町

名古屋大学大学院　人間情報学研究科　認知情報論講座内

Phone: 052-789-4747

Fax: 052-789-4752

jcss@jcss.gr.jp

概 要

認知実験は認知科学の主要な方法の一つであり，そこでは精密な時間制御や測定が必要にされることが多い．本論文ではまず最初に，Windows が導入されているパーソナル・コンピュータ上で動作するタイマの種類と性質について論じた．次にそれらのなかから，コンピュータに内蔵されているタイマ装置を仮想デバイス・ドライバ形式で直接制御するやり方に着目し，Windows 95/98/Me 上で動作するタイマ・ルーチン（wRtcTimr）を作成した。最終的に 10,000 回測定を繰り返すことによって，タイマの精度を検討した．

1. はじめに

パーソナル・コンピュータは実験装置として，今日では認知実験に欠かすことのできないものとなっている．結局ところパーソナル・コンピュータを使って認知実験を行う利点は，精密なやり方で実験手続きを制御できることにある．コンピュータは人間であれば間違いを起こしかねない複雑な手続きを正確に，かつ短時間で実施することができ，しかもそれを何度でも繰り返すことができる．また精度の高いタイマを利用することで，正確な時間スケジュールに沿って刺激を提示したり，反応を取得したりもできるようになる．さらに手続きの進行をコンピュータに委ねることで，実験者効果が結果に現れるのをある程度抑えることができるという利点もある．

本論文ではコンピュータを使用して実験を行う際に，どのように時間制御や測定を行うかに焦点を当て，議論を行うことにする．厳密な時間制御や測定の実現は精度の高い実験を行う上で欠かすことのできないものである．本論文では最初にタイマを作成する三つの方法を取り上げ，その概要について論じる．さらに本論文では，Windows 95/98/Me が導入されているコンピュータで動作するタイマ・ルーチンを作成し，その精度の検討も行う．

2. タイマ実装の方法

実験の性質によっては 1 ミリ秒単位での時間制御・時間測定が求められることがある．パーソナル・コンピュータを使ってこうした微細な単位で時間制御や測定を行う方法はこれまでにも数多く紹介されてきた（e.g., Bührer, Sparrer, & Weitkunat, 1987; Creeger, Miller, & Paredes, 1990; 舟川 , 1988; Kieley, & Higgins, 1989; 下木戸 , 2001a; Westall, Perkry, & Chute, 1986）．これらのタイマはいったいどのようにしてパーソナル・コンピュータに実装されるのだろうか．それにはタイマ装置をコンピュータとは別に用意するか，あるいはコンピュータに備わっている装置を利用するかのいずれかの方法が採られる．コンピュータ内部の装置を利用するとしても装置を直接制御するか，それとも制御を OS（Operating System）に委ねるかによって，この方法はさらに二つに区分することができる．以下にこれらの方法の概要を簡潔に述べ，さらにこれらの方法の利点と問題点についても論じることにする．

2.1. 専用装置の導入

概要　この方法は高周波のクロック生成機能を持つ装置を導入し，これをタイマとして利用するというものである．こうしたタイマ装置はクロック・パルスを生成する水晶発信子と，生成されたパルスを数えるカウンタ，カウンタの値を読み書きしたり，カウンタの動作を制御したりするコントロール・ロジックによって構成され，ISA（Industry Standard Architecture）バス，PCI（Peripheral Component Interconnect）バス, C バスなどに代表される拡張バスや, シリアル・ポートやパラレル・ポートなどに代表される拡張ポートを通してコンピュータと接続することで使用される．利用者はこのコントロール・ロジックを介してタイマ全体の制御を行うことになる．

タイマ装置の基本的な性能はクロック・パルスの周波数とカウンタの大きさによって決定され

る．周波数が高ければ高いほど，より微細な単位での時間制御や測定が可能になり，カウンタの大きさが大きければ大きいほど，タイマ自体が測定できる範囲も大きくなるという性質がある．実際にタイマが扱うことのできる時間範囲はクロック周波数とカウンタの大きさに依存し，例えばクロック周波数が 1 キロ Hz，カウンタの大きさが 16 ビットであるタイマが扱える時間範囲は 1 ミリ秒〜 65.535 秒ということになる．ただしタイマ装置によっては，分周器を使ってクロック周波数をより小さな値に減じているものもある．その場合，クロック周波数とは分周後のクロック周波数のことを指すものとする．

利点と問題点　タイマ装置を導入することの利点は，高精度の時間制御や測定が可能であるという点に尽きる．ソフトウェア・タイマを利用したり，OS が提供する時間関数を利用するというやり方では，厳密な時間制御や測定が必ずしも保証されない．これらの方法はコンピュータ内のタイマ装置を占有できるわけではなく，意図しないタイミングで他のプログラムがタイマ装置にアクセスする場合があるためである．これにより遅延誤差が生じうる．この問題は Windows や Linux のようなマルチ・タスク OS 環境下ではとくに深刻なものとなる．その点では，コンピュータとは別にタイマ専用の装置を導入するやり方はもっとも精度の高いものであるといえる．

逆にこの方法の欠点は，タイマの利用が装置を導入したコンピュータだけに限られるという点である．実験用のコンピュータの分だけ装置を用意しなければならないというのは案外不便なものである．また装置を導入する際のコストも軽視できない．市販されているタイマ・ボードやタイマ装置を利用すると数万円もの費用がかかる．部品を集めてタイマ装置を自作する場合は数千円程度の費用で済むが，この場合は相応の工学的知識が必要になる．

2.2 内部装置の直接制御

概要　PC/AT 互換機と呼ばれるパーソナル・コンピュータには，システム・タイマやリアル・タイム・クロックと呼ばれるタイマ装置がコンピュータの基盤上に既に実装されている．これらの装置を直接制御することでタイマを作成することができる [*1]．

システム・タイマ　PC/AT 互換機にはシステム・タイマと呼ばれる，三組の独立した 16 ビット・カウンタを備えた 8253 互換のチップ・セットが用いられている．これらのタイマはシステムによって予め機能が与えられており，PC/AT 互換機では一つめのもの（カウンタ 0）は CPU へのタイマ割り込み用に，二つめのもの（カウンタ 1）はメモリ（Dynamic Random Access Memory; DRAM）のリフレッシュ（情報の再書き込み）用に，三つめのもの（カウンタ 2）はビープ音の制御用にそれぞれ機能が割り当てられている．利用者がシステム・タイマを直接制御する場合にはカウンタ 0 が選択されることが多い．PC/AT 互換機でのシステム・タイマの基準ク

*1　この他にも CRT（Cathode Ray Tube）ディスプレイが生成する垂直同期信号の周期をタイマとして利用する方法もあり，この方法もしばしば用いられる．

ロック周波数は 1.19318 メガ Hz となっており，利用者はカウンタ 0 に任意の値を書き込むことでこの基準クロックを分周させ，タイマとして利用する．標準では約 55 ミリ秒の周期で CPU に割り込みがかかるように，カウンタ 0 の値が設定されている．

リアル・タイム・クロック　PC/AT 互換機のリアル・タイム・クロックにはモトローラ社の M146818 互換のチップ・セットが用いられている．この装置はカレンダ機能と時計機能を有し，CMOS（Complementary Metal-Oxide Semiconductor）メモリと呼ばれるメモリ領域にシステム構成の情報や日時等のカレンダ情報を保持している．Windows や MS-DOS 等の OS は起動時にこのメモリからカレンダ情報を読み込む．一旦 OS が起動してしまえば日時の管理は OS 自体が行い，以後はほとんど参照されないという性質がある [*2]．そのためリスクを気にせず，気軽に利用できる．このタイマの基準クロック周波数は 32.768 キロ Hz であり，これを分周することで 2 Hz 〜 8.192 キロ Hz となる 13 段階の周波数を生成することができる．標準ではこの周波数は 1.024 キロ Hz に設定されている．

利点と問題点　コンピュータ内部のタイマ装置を直接制御して使用する利点は，特別な装置を用意する必要がないため，低コストで汎用性の高いタイマを利用できるという点である．PC/AT 互換機という同じ仕様のコンピュータを用いる限り，これらのタイマ装置の互換性が保証される．

この方法の問題点は，タイマの利用環境が OS の性質に依存するということである．基本的に Windows のようなマルチ・タスク OS では，コンピュータの資源を利用者が直接制御できない仕様になっている．そのため，Windows 環境でタイマ装置を直接制御するには，制御プログラムをデバイス・ドライバの形式にするという手が考えられる．問題になるのはこのデバイス・ドライバの形式が Windows の種類によって異なることである．VxD 形式と呼ばれるドライバは Windows 95/98/Me で動作するのに対し，WDM（Windows Driver Model）形式と呼ばれるドライバは Windows 98/2000/Me/XP で動作する．両者の間には互換性はないため，VxD 形式で作成されたプログラムは Windows NT/2000/XP では動作しないし，WDM 形式で作成されたプログラムは Windows 95 では動作しない．もう一つの問題は，システム・タイマやリアル・タイム・クロックのクロック周波数に関するものである．これらの装置のクロック周波数はどちらも 1,000 の倍数にはなっていないため，測定値に偏りが生じる．例えば，システム・タイマを 1 ミリ秒単位で使用する場合，タイマの時間周期が 1 ミリ秒に最も近くなるようにカウンタの値を指定するが，厳密に 1 ミリ秒単位で時間測定できるわけではないので，必然的に偏りが生じることになる．PC/AT 互換機の場合，1 ミリ秒あたりにつき約 150.9 ナノ秒の誤差が生じる．数百ミリ秒の時間を扱う場合には，誤差は数十マイクロ秒の単位になる．認知実験ではこの程度の誤差はほとんど問題にならないであろうが，この方法を使用する場合には常にこうした偏りがつきまとうことを念頭に置く必要がある．

---

*2　ただしこのことは MS-DOS や Windows 95/98/Me の OS に限られるかもしれない．Windows NT/2000/XP に関してはこの限りではない．

2.3 API の利用

概要　OS はコンピュータ本体や周辺機器を管理し，それらが持っている機能を統括するものである．OS が有する様々な機能は API（Application Programming Interface）という形式で利用者に提供されており，Windows ではこれらはとくに Win32 API と呼ばれている．これらの関数を呼び出すことで利用者は Windows の持つ様々な機能を利用することができる．こうした API のなかには時間に関する関数も備わっており，GetTickCount，timeGetTime，QueryPerformanceCounter などがそれに該当する．

GetTickCount　Windows 起動時からの経過時間をミリ秒単位で取得する関数であり，32 ビット整数の戻り値を返す．システム・タイマと呼ばれることもある．プロセスの優先度を指定する SetPriorityClass 関数や，指定したミリ秒単位でマルチメディア・タイマを初期化する timeBeginPeriod 関数を併用することによって精度を上げることもできる．ただしその場合でも timeGetTime や QueryPerformanceCounter に比べると精度は落ちる（e.g., 下木戸，2001b）．

timeGetTime　Windows 起動時からの経過時間をミリ秒単位で取得する関数であり，32 ビット整数の戻り値を返す．マルチメディア・タイマと呼ばれることもある．この関数も SetPriorityClass 関数や timeBeginPeriod 関数と併用して，精度を上げることができる．

QueryPerformanceCounter　この関数も Windows 起動時からの経過時間をミリ秒単位で取得するものである．この関数は戻り値ではなく，引数で指定したアドレスに経過時間を 64 ビット整数として返す点が GetTickCount や timeGetTime とは異なる．ここで取り上げた関数の中ではもっとも精度が高い（e.g., 下木戸，2001b）．

利点と問題点　Win32 API を利用することの利点は利便性の高さにある．使用するコンピュータのハードウェア構成や OS，ソフトウェアのバージョン等の違いが OS によって吸収されるため，使用するコンピュータによってハードウェアやソフトウェアを各々用意し，切り替えるという煩わしさがない．OS として Windows が導入されていさえすればどのコンピュータであれば利用できるという利便性の高さは，三つの方法のなかで抜きんでているといえる．

一方，この方法の問題点は，Win32 API のタイマ関数の精度が上で取り上げた方法と比べると落ちることである．さらに問題になるのは，使用するコンピュータの実行環境によってこれらの関数の精度が変わってくることである．例えば，QueryPerformanceCounter 関数は Pentium 以降の CPU（Central Processing Unit）が導入されていないコンピュータでは精度が不明であるし，GetTickCount 関数や timeGetTime 関数は Windows の種類，つまり Windows 95/98/Me を用いるか，それとも Windows NT/2000/XP を用いるかによって精度が異なってくる（e.g., 下木戸，2001b）．さらに，これらの関数はコンピュータ内のどの装置を利用しているのか，対応関係が明確でないという問題もある．GetTickCount 関数はシステム・タイマに，QueryPerformanceCounter 関数は Pentium 以降の CPU に実装されているタイム・スタンプ・カウンタに基づいていることは推察できるものの，それが具体的にどのように対応しているかが明らかにされていないためである．内部仕様を遮蔽する昨今の OS の風潮からしてやむを得ない部

分もあろうが，これらの点が明確になっていない点は利用する際の不安材料になりうる．

3. タイマ・ルーチンの仮想デバイス・ドライバ化

3.1 仮想デバイス・ドライバ

Windows では，利用者がコンピュータ本体や周辺機器を直接制御できない仕様になっているため，MS-DOS 環境で動作するタイマ・ルーチン（e.g., Bührer et al., 1987; Creeger et al., 1990; 舟川 , 1988）のほとんどは Windows 環境では正常に動作しない．そこで考えられるのがタイマ・ルーチンをデバイス・ドライバの形式で記述し，実験プログラムのなかでそれを呼び出すという方法である．タイマ・カウンタのカウントを進め（カウント・アップし）たり，カウントの値を読み込んだり，書き込んだりする処理はタイマの根幹をなすものであるが，それを行うプログラムをデバイス・ドライバの形式で用意し，実験自体を制御するプログラムと分離させればよい．Windows のようなマルチ・タスク OS では，利用者が作成できる各種プログラムの優先度（特権レベル）はある程度のものに抑えられており，もっとも優先度の高い処理はシステム自体（OS）が行うようになっている．プログラムをデバイス・ドライバの形式で記述することで処理の優先度をシステムと同じレベルにできる（Oney, 1996; 大貫，1997）．そうすることにより，タイマ・カウンタの値を読み込んだり，書き込んだりする際に，他のプログラムが割り込んでくることで生じる遅延の影響を最小限に抑えることができる．

Windows では複数のプログラムで一つの装置を共有するため，各装置は Windows システム上で仮想化されており，各プログラムはこの仮想化された装置（Virtual Machine）を介して，実際の装置を制御するようになっている．つまり実際の装置と仮想上の装置を区別することで，複数のプログラムが一つの装置に対して同時にアクセスする際に生じる競合の問題を回避している．プログラムがある装置にアクセスする場合，具体的には以下のような手順をとる．装置が持つ機能を利用するため，まずプログラムはその装置に対してアクセス要求を発信する．実際にはこの要求は，OS によって仮想化された装置に対して送られることになる．OS と装置との情報のやり取りはデバイス・ドライバと呼ばれるプログラムが担当するが，Windows ではこれらはとくに仮想デバイス・ドライバ（Virtual Device Driver）と呼ばれている．仮想デバイス・ドライバは各プログラムから送信された要求を一旦受け取って置き，OS によって CPU 資源が割り当てられた時点ではじめて，実際の装置に対してアクセスを行う．実際の装置にアクセスするタイミングは OS が決定するので，複数のアクセス要求が競合することはない．実際の装置にアクセスした結果は仮想デバイス・ドライバによってもとのプログラムに送られる．これらの処理は自動的に行われるため，プログラムの作成者は実際の装置や仮想化された装置の違いを意識する必要はない．

3.2 タイマ・ルーチン（wRtcTimr）の作成

本論文では，PC/AT 互換機に内蔵されているリアル・タイム・クロックを制御して，1 ミリ秒

単位で動作するタイマ・ルーチン（wRtcTimr）の作成を試みる．wRtcTimr は PC/AT 互換機に内蔵されているリアル・タイム・クロックを制御し，そのタイマ割り込みを利用することでカウント・アップを行う．この処理自体は仮想デバイス・ドライバの形式，なかでも VxD 形式で作成する．そのため wRtcTimr は VxD 形式のドライバが使用される Windows 95/98/Me 環境の下でしか動作しない．Windows 95/98/Me 環境ではリアル・タイム・クロックはあまり利用されていないため，装置を制御する際のリスクをあまり気にしなくてすむ．

wRtcTimr はタイマルーチン本体である wRtcTimr.vxd とそれを利用するための API 関数を提供するファイル群（wRtcTimr.dll，wRtcTimr.h，wRtcTimr.lib）から構成される．wRtcTimr は API 関数を介して利用者に機能を提供する（表 1 参照）．wRtcTimr.dll は wRtcTimr.vxd の機能を API 関数として利用できるように表現し直したものである．実際には，プログラム中で wRtcTimr.h ファイルをインクルードし，リンク時に wRtcTimr.lib をリンクすることで利用可能になる．

wRtcTimr.vxd のプログラムはアセンブラで作成し，wRtcTimr.dll は C 言語で作成した（付録参照）．VxD 形式の仮想デバイス・ドライバは普通のソフトウェアでは作成できないため，Microsoft 社が提供している Windows 98 DDK（Device driver Development Kit）に含まれるツール（nmake.exe，ml.exe，link.exe，mapsym.exe）を用いた．Windows 98 DDK に含まれている setenv.bat を実行し，DDK の環境変数を設定した後，nmake.exe を使って wRtcTimr.vxd を作成した．

3.3 wRtcTimr の性質

wRtcTimr は PC/AT 互換機に内蔵されているリアル・タイム・クロックを用いており，0.9765625 ミリ秒ごとに割り込み（ハードウェア割り込み）を発してカウントを進めている．そのため，このカウンタ値に基づいて時間制御や測定を行う場合，1 カウントあたり 23.4375 マイクロ秒の誤差が生じる．この誤差は数百ミリ秒以上の時間を扱う場合には数ミリ秒の単位になるので，実際に使用する際には注意が必要である．wRtcTimr のカウンタ値と実際の時間にはカウンタ値／実際の時間＝ 1.024 の関係があり，これを利用して補正（較正）を行うことができる．例えば wRtcTimr を用いて 500 ミリ秒待機する場合，wRtcTimr のカウンタに 500 を指定すると，実際には 488.28125 ミリ秒分しか待機しない．意図通り 500 ミリ秒待機させようとする場合には，500 に 1.024 を乗じた値，512 をカウンタに指定してやればよい．

4. 精度の検討

4.1 タイマの精度

タイマを用いて時間制御や測定を行う場合，ある程度の誤差が混入するのは現実的に避けられないことである．このような誤差は系統誤差と偶然誤差に分類することができる．系統誤差は偏りを持った誤差であり，タイマの癖のような要因が原因となって生じる．一方，偶然誤差は測定

表 1　wRtcTimr が提供する API

関数名	機能
BOOL RtctimerVersion(HANDLE hVxD, LPWORD lpVerNo);	タイマのバージョンを取得する
BOOL RtctimerOpen(LPHANDLE lpVxD);	タイマをオープンする（必須）
BOOL RtctimerClose(HANDLE hVxD);	タイマをクローズする（必須）
BOOL RtctimerStart(HANDLE hVxD);	カウント・アップを開始する
BOOL RtctimerStop(HANDLE hVxD);	カウント・アップを終了する
BOOL RtctimerResetCount(HANDLE hVxD);	カウント値をリセットする
BOOL RtctimerGetCount(HANDLE hVxD, LPDWORD lpCnt);	カウント値を取得する
BOOL RtctimerWaitInterval(HANDLE hVxD, LPDWORD lpTim);	指定された時間分待機する

を行う度にばらつきとして現れる誤差であり，測定時の電圧の変化といった測定条件の変動などの原因によって生じる．こうした誤差は時間制御や測定の際の精度（accuracy）と密接な関係があり，系統誤差は正確さ（trueness）と，偶然誤差は精密さ（precision）と対応している．

正確さは繰り返し時間測定を行った場合に，本来測ろうとした値からどの程度測定値が偏っているのかを示すものであり，値が高いほど意図したとおりの時間制御や測定が可能であることを意味する．一方，精密さはタイマを用いて繰り返し時間測定を行った場合に，測定値がどの程度ばらついているかを示すものであり，値が高いほど安定した時間制御や測定が可能であることを意味する．精度は正確さと精密さを総合した信頼性の指標である．

正確さは正確率によって，精密さは精密度によって評価することができる．正確率は産出された平均から真の値を減じたものと真の値との比率によって定義され，測定時の偏りが真の値を基準とした際にどれだけの大きさになるのかを表している．一方，精密度は産出された標準偏差と産出された平均との比率によって定義され，測定時のばらつきがどのくらいの大きさになるのかを平均を基準として示している．これは一般には変動係数として知られている．本論文ではこれらの指標を用いて wRtcTimr の精度の検討を行う．

4.2 本論文の目的

本論文では wRtcTimr が Windows 上で実際にどのような精度で動作するのか，Interface 社のタイマ・ボード，PCI-6103 を用いて評価する．具体的には wRtcTimr の RtctimerWaitInterval 関数で 500 ミリ秒待機した際に，PCI-6103 のカウンタ値がどのような値を示すのかを検討する．PCI-6103 は 1 メガ Hz のクロック周波数と 32 ビットのカウンタを備えたタイマ・ボードであり，フリーラン・カウンタ機能を用いれば 1 マイクロ秒の単位で最大 1 時間 19 分（2^{32} — 1 マイクロ秒）までの時間測定が可能である．さらに本論文では，Win32 API の時間関数である GetTickCount，timeGetTime，QueryPerformanceCounter の精度も評価し，wRtcTimr との比較も行う．

4.2.1 方 法

装 置　使用したコンピュータは Dell OptiPlex GXa であり，CPU が Pentium Ⅱの 266 メガ Hz，OS は Windows 95（version 4.00.950b）であった．タイマの精度の評価には Interface 社製タイマ・ボード PCI-6103 を用いた．

手続き　本論文では wRtcTimr が実際にどれくらいの精度を有するのかを評価するため，wRtcTimr で 500 カウント分待機したときの PCI-6103 のカウンタ値を測定した．この評価は 10,000 回分行われた．評価のためのプログラムは Win32 API の SetPriorityClass 関数を用いてプロセスの優先度が設定され，具体的にはタイム・クリティカルなもの（HIGH_PRIORITY_CLASS）になるように指定した．

4.2.2 結果と考察

精度評価の結果を表 2 に示す．wRtcTimr で 500 カウント分待機した際に期待されるカウント値は 488,281.25 である．これは wRtcTimr のクロック周波数が 1,024 Hz であり，1 カウントに要する時間が実際には 0.9765625 ミリ秒となっているためである．10,000 回にわたる測定の結果，実際に得られた測定値の平均は 490.7 ミリ秒であった．この値は期待された値よりも 2.42 ミリ秒遅いものである．標準偏差は 3.99 ミリ秒であり，パーセンタイルを見ても 90 パーセントのデータが 1 標準偏差以内に収まっている．これらの点を考慮すると wRtcTimr の精度は数ミリ秒であると考えられる．

一方，Win32 API の時間関数の精度は概ね wRtcTimr よりも上回っていた．実際に得られた測定値の平均は GetTickCount で 508.2 ミリ秒，timeGetTime と QueryPerformanceCounter で 500.0 ミリ秒であった．標準偏差は GetTickCount で 1.89 ミリ秒，timeGetTime で 1.13 ミリ秒，QueryPerformanceCounter で 2.29 ミリ秒であった．とくに timeGetTime と QueryPerformanceCounter は 95 パーセントのデータが平均から 1 ミリ秒以内に収まっており，1 ミリ秒の精度をもっているといえよう．

もっとも精度がよくないとされる GetTickCount の正確率が wRtcTimr より劣っていた点を除けば，正確率，精密度のいずれも Win32 API の時間関数の方が wRtcTimr よりも値がよかった．この理由としてまず一つには，測定に使用したコンピュータの OS が Windows 95 であった点が挙げられる．Windows 95/98/Me の Win32 API の時間関数の精度は高く，適切な指定を行っていれば大体 1 ミリ秒程度の精度をもつことが報じられている（e.g., 下木戸，2001b）．次に，wRtcTimr の割り込み処理に関するオーバーヘッドが考えていた以上に大きかった点が考えられる．wRtcTimr ではリアル・タイム・クロックからハードウェア割り込みがかかる度に，カウンタの値を進め，リアル・タイム・クロックのレジスタを読み込み，次の割り込みに備えるという手続きを行っている．OS がハードウェア割り込みを検出し，wRtcTimr の割り込み処理が呼び出される手続きに思いの外，時間を要したため，それが遅延となって測定値に反映されたと考えられる．コンピュータの CPU が高速なものであるほどこの遅延は小さくなるので，より高性能の CPU をもつコンピュータで測定を行えば wRtcTimr の精度は向上する可能性がある．

結論を述べると，wRtcTimr の精度は認知実験ではほぼ充分な精度をもっているといえる

表２　wRtcTimr（wRT），GetTickCount（GTC），timeGetTime（tGT），
QueryPerformanceCounter（QPC）のそれぞれで 500 カウント分待機したときの実測値の平均値，
標準偏差（SD），パーセンタイル，正確率（%）および精密度（%）

条件	平均	SD	最小	1%	5%	10%	25%	50%	75%	90%	95%	99%	最大	正確率	精密度
wRT	490.7	3.99	487.8	488.3	488.6	488.8	489.2	489.8	490.6	492.4	494.6	510.2	551.3	0.50	0.82
GTC	508.2	1.89	496.8	505.9	507.5	507.7	507.9	508.0	508.2	508.4	508.6	513.0	558.6	1.63	0.37
tGT	500.0	1.13	498.9	499.4	499.8	499.9	499.9	499.9	499.9	499.9	500.0	501.9	535.3	0.00	0.22
QPC	500.0	2.29	499.8	499.9	499.9	499.9	499.9	499.9	499.9	499.9	499.9	501.1	696.1	0.00	0.46

注：測定はそれぞれ 10,000 回行った.

が，より利便性の高い Win32 API の時間関数の方が精度が高かった点を考えると，このタイマを導入する利点は現状ではほとんどないといわざるを得ない．Windows NT/2000/XP 環境下ではこれらの時間関数の精度がやや落ちることが報じられているため（e.g., 下木戸，2001b），wRtcTimr が有用であるとしたら，これらの OS 環境の下であるかもしれない．本論文で作成した wRtcTimr は Windows NT/2000/XP では動作しないが，wRtcTimr を WDM 形式で作成し，これらの OS の下での精度を検討することが必要であろう.

5. 討　論

刺激の作成や，実験手続きの制御，データの取得やその分析に渡る一連の研究過程において，今日ではパーソナル・コンピュータは欠かせないものとなっている．Psychology Software Tools 社の MEL（Micro Experimental Laboratory）や E-Prime に代表される認知実験ソフトウェアは，インターフェースのなじみ易さや利便性の高さから急速に普及してきている．しかしながら，これらの実験ソフトウェアは時間制御や測定を何に基づいて行っているのかが明らかになっていないため，時間制御に実際どのくらいの精度を有しているのかが不明である．利用者自身がプログラム言語で心理学実験を作成する場合は，特別な装置や OS，ソフトウェア（タイマ･ルーチン）が提供するタイマ機能を利用することになるが，これらの場合は利用者が必要に応じてそのタイマ機能の精度を評価することができるため，実験の目的や性質に適った精度をもつタイマを選択できる．この利点は決して少なくないと思われる．今後は，コンピュータや心理学実験にあまりなじみのない初学者が気軽に心理学実験プログラムを作成することができるような，プログラム環境やツールを整備し，構築していくことが必要となろう．そうした試みの一環として，精度が明らかにされているタイマ・ルーチンを公表することは意義があるといえる.

文 献

Bührer, M., Sparrer, B., & Weitkunat, R. (1987). Interval timing routines for the IBM PC/XT/AT microcomputer family. *Behavior Research Methods, Instruments, & Computers*, **19**, 327-334.

Creeger, C. P., Miller, K. F., & Paredes, D. R (1990). Micromanaging time: Measuring and controlling timing errors in computer-controlled experiments. *Behavior Research Methods, Instruments, & Computers*, **22**, 34-79.

舟川政美 (1988). MASM によるプログラミングと実験制御 . 阿部純一（編）,『パーソナル・コンピュータによる心理学実験プログラミング』. ブレーン出版 .

Kieley, J. M., & Higgins, T. S. (1989). Precision timing options for the Apple Macintosh family of computers. *Behavior Research Methods, Instruments, & Computers*, **21**, 259-264.

Oney, W. (1996). *Systems programming for Windows 95.* Redmond, WA.: Microsoft Press.（太田博志 他訳 (1997).『Windows95 システムプログラム開発』. アスキー出版 .)

大貫広幸 (1997).『仮想デバイスドライバの作り方：Windows95 の周辺機器を動かすための基礎知識』. CQ 出版社 .

下木戸隆司 (2001a). 心理学実験における MS-DOS ―ミリ秒単位のタイマ･ルーチン DosTimer を中心として― .『名古屋大学大学院教育発達科学研究科紀要（心理発達科学）』, **48**, 315-341.

下木戸隆司 (2001b). Win32 API を利用した時間計測と制御の問題―タイマ関連 API の精度と安定性― .『教育心理学論集（名古屋大学教育学研究科）』, **30**, 51-59.

Westall, R., Perkey, M. N., & Chute, D. L. (1986). Accurate millisecond timing on Apple's Macintosh using Drexel's MilliTimer. *Behavior Research Methods, Instruments, & Computers*, **18**, 307-311.

付録 1　wRtcTimr のソース（wrtctimr.asm）

```
        page    60,132

        title   WRTCTIMR.ASM Ver 1.0

.386p

;-------------------------------------------------------;
;            RTC Timer for Windows VxD                  ;
;                    Ver 1.0                              ;
;-------------------------------------------------------;

.XLIST

WIN40COMPAT
include  VMM.INC
include  DEBUG.INC
include  VPICD.INC
include  WIN32.INC
include  WINERROR.INC
include  BASEDEF.INC

.LIST

WRTCIRQ = 08h
RTCCntrl = 0070h
RTCIC = 0071h

VPICDOpt = 0003h                        ; VPICD_OPT_READ_HW_IRR OR VPICD_OPT_CAN_SHARE

; WRTCTIMR Version No. の定義
VER_maj = 1
VER_min = 0
VER = VER_maj*100h+VER_min

; clc 命令 + ret 命令
clcret macro
        clc
        ret
endm

; stc 命令 + ret 命令
stcret macro
        stc
        ret
endm
```

```
;-------------------------------------------------------------;
;                仮想デバイス WRTCTIMR の宣言                 ;
;-------------------------------------------------------------;
Declare_Virtual_Device WRTCTIMR, Ver_maj, Ver_min, CtrlProc, Undefined_Device_ID, , ,

;-------------------------------------------------------------;
;                      データ・セグメント                     ;
;-------------------------------------------------------------;
VxD_LOCKED_DATA_SEG

        public  WRTCTIM_Desc
WRTCTIM_Desc VPICD_IRQ_Descriptor <WRTCIRQ, VPICDOpt, \
                                   OFFSET32 VIRQ_Hw_INT,      \
                                   OFFSET32 VIRQ_Virt_INT,    \
                                   OFFSET32 VIRQ_EOI,         \
                                   OFFSET32 VIRQ_Mask_Change, \
                                   OFFSET32 VIRQ_IRET>

        public  hRTC_IRQ_Handle
hRTC_IRQ_Handle  dd      ?       ; IRQ のﾊﾝﾄﾞﾗ

        public  RtcOpenFlag, hRtcOpenDevice
RtcOpenFlag      dw      FALSE
hRtcOpenDevice   dd      0

        public  dwRTC_Count
dwRTC_Count      dd      0

        public  byOrgRTC0aReg, byOrgRTC0bReg
byOrgRTC0aReg    db      ?
byOrgRTC0bReg    db      ?

w32_procTable label dword
        dd      OFFSET32 RtcVersion
        dd      OFFSET32 RtcOpen
        dd      OFFSET32 RtcClose
        dd      OFFSET32 RtcStart
        dd      OFFSET32 RtcStop
        dd      OFFSET32 RtcResetCount
        dd      OFFSET32 RtcGetCount
        dd      OFFSET32 RtcWaitInterval

w32_procMax      equ     ($-w32_procTable)/4

VxD_LOCKED_DATA_ENDS

;-------------------------------------------------------------;
;                      コード・セグメント                     ;
;-------------------------------------------------------------;
VxD_LOCKED_CODE_SEG
;
;   VxD ｺﾝﾄﾛｰﾙ･ﾌﾟﾛｼｰｼﾞｬ
;
```

```
BeginProc CtrlProc
        Control_Dispatch SYS_DYNAMIC_DEVICE_INIT, dynDevInit
        Control_Dispatch SYS_DYNAMIC_DEVICE_INIT, dynDevExit
        Control_Dispatch W32_DEVICEIOCONTROL, IOctrl
        clcret
EndProc CtrlProc

;
;
;
BeginProc dynDevInit
        clcret
EndProc dynDevInit

;
;
;
BeginProc dynDevExit
        clcret
EndProc dynDevExit

;
;
;
BeginProc IOctrl
        cmp     ecx, DIOC_OPEN                          ; オープン済みか？
        jz      IOctrl_ok_ret
        cmp     ecx, DIOC_CLOSEHANDLE                   ; クローズしているか？
        jz      RtcClose
        dec     ecx
        cmp     ecx, w32_procMax                        ; サービス番号を調べる
        jae     IOctrl_ns_error_ret                     ; エラーは発生しているか？
        call    w32_procTable[ecx*4]                    ; 提供処理を実行
        jc      IOctrl_error_ret                        ; エラーは？

IOctrl_ok_ret:                                          ; 成功
        xor     eax, eax                                ; EAX=0, CF=0
        clcret
IOctrl_ns_error_ret:                                    ; サービス番号が正しくない
        mov     eax, ERROR_NOT_SUPPORTED
IOctrl_error_ret:                                       ; 失敗
        stcret                                          ; EAX=error code, CF=1
EndProc IOctrl

;
;
;
BeginProc RtcOpen
        cmp     RtcOpenFlag, TRUE                       ; オープン済みか？
        jz      error_acc


        pushf
        cli
        mov     al, 0ah
```

```
        mov     dx, RTCCntrl
        out     dx, al
        IO_DELAY
        IO_DELAY
        mov     dx, RTCIC
        in      al, dx
        mov     byOrgRTC0aReg, al                       ; 0a レジスタの内容を保存

        and     al, 0f0h                                ; Oa レジスタの下位 4 ビットを
        or      al, 06h                                 ; 0110 にセット，上位はそのまま
        mov     dx, RTCIC
        out     dx, al

        mov     al, 0bh
        mov     dx, RTCCntrl
        out     dx, al
        IO_DELAY
        IO_DELAY
        mov     dx, RTCIC
        in      al, dx
        mov     byOrgRTC0bReg, al                       ; 0b レジスタの内容を保存

        and     al, 8fh                                 ; 0b レジスタの 6-4 ビットを
        or      al, 40h                                 ; 100 にセット，残りはそのまま
        mov     dx, RTCIC
        out     dx, al
        mov     edi, OFFSET32 WRTCTIM_Desc
        VxDCall VPICD_Virtualize_IRQ                    ; 仮想割り込みを登録
        jc      irq_inst_err
        mov     hRTC_IRQ_Handle, eax

        mov     eax, hRTC_IRQ_Handle
        VxDCall VPICD_Physically_mask
        popf

        mov     eax, [esi].hDevice
        mov     hRtcOpenDevice, eax
        mov     RtcOpenFlag, TRUE

        clcret
 irq_inst_err:
        stcret
EndProc RtcOpen

;
;  エラー終了
;
        public  error_len, error_ptr
error_len:      ; IOCTL のバッファの長さが正しくない
        stcret
error_ptr:      ; IOCTL のバッファを示すポインタが正しくない
        stcret
error_acc:      ; アクセスできない
        stcret
;
```

```
;   WRTCTIMR の Version No. の取得
;
BeginProc RtcVersion
        cmp     [esi].cbOutBuffer, type WORD
        jb      error_len
        mov     ebx, [esi].lpvOutBuffer
        or      ebx, ebx
        jz      error_ptr
        mov     WORD ptr [ebx], VER                             ; Version No. の設定
        mov     ebx,[esi]. lpcbBytesReturned
        mov     dword ptr [ebx], type WORD

        clcret
EndProc RtcVersion

;
;
;
BeginProc RtcClose
        cmp     RtcOpenFlag, FALSE                              ; オープン済みか？
        jz      error_acc
        mov     eax, [esi].hDevice
        cmp     eax, hRtcOpenDevice                             ; オープンしたデバイスか？
        jnz     error_acc
;
        pushf
        cli
        mov     al, 0ah
        mov     dx, RTCCntrl
        out     dx, al
        mov     dx, RTCIC
        out     dx, al

        mov     al, 0bh
        mov     dx, RTCCntrl
        out     dx, al                                          ; 0b レジスタの内容の復帰
        mov     al, byOrgRTC0bReg
        mov     dx, RTCIC
        out     dx, al

        mov     eax, hRTC_IRQ_Handle
        VxDCall         VPICD_Set_Auto_Masking
        mov     eax, hRTC_IRQ_Handle
        VxDCall         VPICD_Force_Default_Behavior            ; 登録した割り込みを削除

        mov     RtcOpenFlag, FALSE
        popf
        jmp     IOctrl_ok_ret
EndProc RtcClose

;
;
;
```

```
BeginProc RtcStart
        cmp     RtcOpenFlag, FALSE                      ; オープン済みか？
        jz      IOctrl_ok_ret
        mov     eax, [esi].hDevice
        cmp     eax, hRtcOpenDevice                     ; オープンしたデバイスか？
        jnz     IOctrl_ok_ret

        mov     eax, hRTC_IRQ_Handle
        VxDCall VPICD_Physically_Unmask

        clcret
EndProc RtcStart

;
;
;
BeginProc RtcStop
        cmp     RtcOpenFlag, FALSE                      ; オープン済みか？
        jz      IOctrl_ok_ret
        mov     eax, [esi].hDevice
        cmp     eax, hRtcOpenDevice                     ; オープンしたデバイスか？
        jnz     IOctrl_ok_ret

        mov     eax, hRTC_IRQ_Handle
        VxDCall VPICD_Physically_Mask

        clcret
EndProc RtcStop

;
;
;
BeginProc RtcResetCount, HIGH_FREQ
        cmp     RtcOpenFlag, FALSE                      ; オープン済みか？
        jz      IOctrl_ok_ret
        mov     eax, [esi].hDevice
        cmp     eax, hRtcOpenDevice                     ; オープンしたデバイスか？
        jnz     IOctrl_ok_ret

        mov     dwRTC_Count, 0
        clcret
EndProc RtcResetCount

;
;
;
BeginProc RtcGetCount, HIGH_FREQ
        cmp     RtcOpenFlag, FALSE                      ; オープン済みか？
        jz      IOctrl_ok_ret
        mov     eax, [esi].hDevice
        cmp     eax, hRtcOpenDevice                     ; オープンしたデバイスか？
        jnz     IOctrl_ok_ret

        cmp     [esi].cbOutBuffer, type dword
```

```
        jb      error_len
        mov     ebx, [esi].lpvOutBuffer
        or      ebx, ebx
        jz      error_ptr

        mov     eax, dwRTC_Count
        mov     dword ptr [ebx], eax
        mov     ebx, [esi].lpcbBytesReturned
        mov     dword ptr [ebx], type dword

        clcret
EndProc RtcGetCount

;
;
;
BeginProc RtcWaitInterval, HIGH_FREQ
        cmp     RtcOpenFlag, FALSE                      ; オープン済みか？
        jz      IOctrl_ok_ret
        mov     eax, [esi].hDevice
        cmp     eax, hRtcOpenDevice                     ; オープンしたデバイスか？
        jnz     IOctrl_ok_ret

        cmp     [esi].cbInBuffer, type dword
        jb      error_len
        mov     ebx, [esi].lpvInBuffer
        or      ebx, ebx
        jz      error_ptr
        mov     eax, dword ptr [ebx]

        or      eax, eax
        jz      exit_wait_loop
        mov     dwRTC_Count, 0

 wait_loop:
        cmp     eax, dwRTC_Count
        jnz     wait_loop

 exit_wait_loop:
        mov     dwRTC_Count, 0

        clcret
EndProc RtcWaitInterval

;
;   VID_Hw_Int_Proc（物理ハードウェア割り込みの処理）
;
BeginProc VIRQ_Hw_Int, HIGH_FREQ

        add     dwRTC_Count, 1

        mov     al, 0ch                                 ; 次の割り込みを可能にするため
        mov     dx, RTCCntrl
        out     dx, al                                  ; 0c レジスタを読み込む
```

```
        mov     dx, RTCIC
        in      al, dx

        mov     eax, hRTC_IRQ_Handle
        VxDCall VPICD_Phys_EOI
        clcret
EndProc VIRQ_Hw_Int

;
;   VID_Virt_Int_Proc（仮想ハードウェア割り込み時のコールバック）
;
BeginProc VIRQ_Virt_Int, HIGH_FREQ
        ret
EndProc VIRQ_Virt_Int

;
;   VID_EOI_Proc（仮想 PIC に EOI が発行された時のコールバック）
;
BeginProc VIRQ_EOI, HIGH_FREQ
        VxDCall VPICD_Phys_EOI
        VxDJmp  VPICD_Clear_Int_Request
EndProc VIRQ_EOI

;
;   VID_IRET_Proc（仮想マシンで IRET 命令実行時のコールバック）
;
BeginProc VIRQ_IRET, HIGH_FREQ
        ret
EndProc VIRQ_IRET

;
;   VID_Mask_Change_proc（仮想 PIC の割り込みマスクが変更された時のコールバック）
;
BeginProc VIRQ_Mask_Change, HIGH_FREQ
        ret
EndProc VIRQ_Mask_Change

VxD_LOCKED_CODE_ENDS

        END
```

付録 2　wRtcTimr.c のソース

```
/*
    WRTCTIMR.VXD を呼び出すための API の定義

    ｺﾝﾊﾟｲﾙ方法（CL 使用）
      cl /LD wrtctimr.lib
*/

#include <string.h>
#include <stdlib.h>
#include <windows.h>
#include <vmm.h>

#define W32_RtcVersion       (1)   // WRTCTIMR.VXD のバージョン番号の取得
  // InBuffer  : なし
  // OutBuffer : WORD --> バージョン番号

#define W32_RtcOpen          (2)   // タイマ・オープン
  // InBuffer  : なし
  // OutBuffer : なし

#define W32_RtcClose         (3)   // タイマ・クローズ
  // InBuffer  : なし
  // OutBuffer : なし

#define W32_RtcStart         (4)   // 割り込み開始
  // InBuffer  : なし
  // OutBuffer : なし

#define W32_RtcStop          (5)   // 割り込み終了
  // InBuffer  : なし
  // OutBuffer : なし

#define W32_RtcResetCount    (6)   // カウント値リセット
  // InBuffer  : なし
  // OutBuffer : なし

#define W32_RtcGetCount      (7)   // カウント値取得
  // InBuffer  : なし
  // OutBuffer : DWORD --> カウンタ値

#define W32_RtcWaitInterval  (8)   // 指定時間待機
  // InBuffer  : DWORD --> 待機時間
  // OutBuffer : なし

DWORD cbBytesReturned;   /* DeviceIoControl から返された */
                         /* outBuffer への書込ﾊﾞｲﾄ数     */

/* WRTCTIMR.VXD のバージョン番号取得 */
__declspec(dllexport) BOOL RtctimerVersion(HANDLE hVxD, LPWORD lpVerNo);
```

```
/* タイマ・オープン */
__declspec(dllexport) BOOL RtctimerOpen(void);

/* タイマ・クローズ */
__declspec(dllexport) BOOL RtctimerClose(HANDLE hVxD);

/* 割り込み開始 */
__declspec(dllexport) BOOL RtctimerStart(HANDLE hVxD);

/* 割り込み終了 */
__declspec(dllexport) BOOL RtctimerStop(HANDLE hVxD);

/* カウント値リセット */
__declspec(dllexport) BOOL RtctimerResetCount(HANDLE hVxD);

/* カウント値取得 */
__declspec(dllexport) BOOL RtctimerGetCount(HANDLE hVxD, LPDWORD lpCnt);

/* 指定時間待機 */
__declspec(dllexport) BOOL RtctimerWaitInterval(HANDLE hVxD, LPDWORD lpTim);


/* VxD WRTCTIMR.VXD のﾊﾞｰｼﾞｮﾝ番号 */
BOOL RtctimerVersion(HANDLE hVxD, LPWORD lpVerNo) {
    return DeviceIoControl(hVxD, W32_RtcVersion,
             NULL, 0,
             lpVerNo, sizeof(*lpVerNo),
             &cbBytesReturned, NULL);
}

/* タイマ・オープン */
HANDLE RtctimerOpen(void) {
    HANDLE hVxD;
    hVxD =CreateFile("\\\\.\\WRTCTIMR.VXD", 0, 0, NULL, 0,
             FILE_FLAG_DELETE_ON_CLOSE, NULL);
    DeviceIoControl(hVxD, W32_RtcOpen,
        NULL, 0,
        NULL, 0,
        &cbBytesReturned, NULL);
    return hVxD
}

/* タイマ・クローズ */
BOOL RtctimerClose(HANDLE hVxD) {
    DWORD dwRc;
    dwRc = DeviceIoControl(hVxD, W32_RtcClose,
             NULL, 0,
             NULL, 0,
             &cbBytesReturned, NULL);
    if (!dwRc) {
        return dwRc;
    }
    return CloseHandle(hVxD);
}
```

```
/* 割り込み開始 */
BOOL RtctimerStart(HANDLE hVxD) {
    return DeviceIoControl(hVxD, W32_RtcStart,
             NULL, 0,
             NULL, 0,
             &cbBytesReturned, NULL);
}

/* 割り込み終了 */
BOOL RtctimerStop(HANDLE hVxD) {
    return DeviceIoControl(hVxD, W32_RtcStop,
             NULL, 0,
             NULL, 0,
             &cbBytesReturned, NULL);
}

/* カウント値リセット */
BOOL RtctimerResetCount(HANDLE hVxD) {
    return DeviceIoControl(hVxD, W32_RtcResetCount,
             NULL, 0,
             NULL, 0,
             &cbBytesReturned, NULL);
}

/* カウント値取得 */
BOOL RtctimerGetCount(HANDLE hVxD, LPDWORD lpCnt) {
    return DeviceIoControl(hVxD, W32_RtcGetCount,
             NULL, 0,
             lpCnt, sizeof(*lpCnt),
             &cbBytesReturned, NULL);
}

/* 指定時間待機 */
BOOL RtctimerWaitInterval(HANDLE hVxD, LPDWORD lpTim) {
    return DeviceIoControl(hVxD, W32_RtcWaitInterval,
             lpTim, sizeof(*lpTim),
             NULL, 0,
             &cbBytesReturned, NULL);

}
```

付録 3　wRtcTimr.h のソース

```
/*
    WRTCTIMR.VXD を呼び出すためのヘッダファイル
*/

#ifndef __WRTCTIMR_H
#define __WRTCTIMR_H

#include <windows.h>
#include <vmm.h>



/* VxD WRTCTIMR.VXD のバージョン番号 */
extern __declspec(dllexport) BOOL RtctimerVersion(HANDLE hVxD, LPWORD lpVerNo);

/* タイマ・オープン */
extern __declspec(dllexport) BOOL RtctimerOpen(void);

/* タイマ・クローズ */
extern __declspec(dllexport) BOOL RtctimerClose(HANDLE hVxD);

/* 割り込み開始 */
extern __declspec(dllexport) BOOL RtctimerStart(HANDLE hVxD);

/* 割り込み終了 */
extern __declspec(dllexport) BOOL RtctimerStop(HANDLE hVxD);

/* カウント値リセット */
extern __declspec(dllexport) BOOL RtctimerResetCount(HANDLE hVxD);

/* カウント値取得 */
extern __declspec(dllexport) BOOL RtctimerGetCount(HANDLE hVxD, LPDWORD lpCnt);

/* 指定時間待機 */
extern __declspec(dllexport) BOOL RtctimerWaitInterval(HANDLE hVxD, LPDWORD lpTim);

#endif /* __WRTCTIMR_H */
```