CHAPTER 8

CSS スクリプト言語の使用方法

CSS は、日常業務の手順を自動化するスクリプトを作成できる充実したスクリプト言語を備えています。スクリプト言語は、スクリプト キープアライブで使用するスクリプトを、お客様独自のサービス要件に合わせて作成する手段として特に役立ちます。スクリプト キープアライブの詳細については、『*Cisco Content Services Switch Content Load-Balancing Configuration Guide*』を参照してください。

（注） スクリプト例に含まれるコマンドは、分かりやすいように太字で示しています。

スクリプトでは、Command Line Interface（CLI; コマンド行インターフェイス）の任意のコマンドを使用することができます。スクリプトでは、条件や式に基づいて特定のコマンドを実行する論理ブロックも利用できます。

この章の主な内容は次のとおりです。

スクリプトに関する留意事項

CSS ソフトウェアをアップグレードすると、古いスクリプトは旧バージョンのソフトウェアの /script ディレクトリに残ります。旧バージョンのソフトウェアのディレクトリから新しいソフトウェアのディレクトリにスクリプトをコピーする方法については、この章で後述する「スクリプトのアップグレードにあたっての留意事項」を参照してください。

CSS スクリプト言語では、引用符で囲まれた 128 文字の引数を渡すことができます。1 つの引数につき平均 7 文字（およびスペース区切り）とすると、1 つのスクリプトで最大 16 の引数を使用できます。

スクリプトの実行

スクリプトはローカルのハード ディスクまたはフラッシュ ディスクの /script ディレクトリから実行されます。このディレクトリに保存したスクリプトだけが、**script play** コマンドにアクセスできます。

スーパーユーザ モードで CLI からスクリプトを 1 行ずつ実行するには、**script play** コマンドを使用します。このコマンドを使用して、スクリプトにパラメータ値を渡すこともできます。

たとえば、次のように入力します。

```
# script play MyScript "Argument1 Argument2"
```

利用可能なスクリプトのリストは、**show script** コマンドを使用して表示できます。

コマンド スケジューラの使用方法

定期的なコンテンツの複製、統計情報の集計、および定期的な設定変更を行う CLI コマンドの実行をスケジューリングすることができます。スクリプトの実行などの、CLI コマンドの実行スケジュールを設定するには、**cmd-sched** コマンドを使用します。実行されるコマンドは、コマンド文字列と呼ばれます。コマンドをスケジューリングするには、設定レコードを作成する必要があります。このレコードには、コマンド文字列とコマンドの実行タイミングに関する規定が含まれています。

コマンド スケジューラは、各文字列が実行される擬似ログイン シェルを作成して、指定した時間にコマンド文字列を実行します。**cmd-sched** レコードは、そのシェルの終了時だけに実行スケジュールが作成されます。アクティブな擬似シェルの情報を表示するには、**show lines** コマンドを使用します。

（注）　コマンド文字列の実行を終了するには、**disconnect** コマンドを使用します。

このグローバル設定モード コマンドのシンタックスとオプションは次のとおりです。

- **cmd-sched**：コマンドのスケジュールを有効にする。
- **cmd-sched record** *name minute hour day month weekday* "*commands...*" {*logfile_name*}：スクリプトの実行など、スケジュールされた CLI コマンドの実行のための設定レコードを作成する。

このコマンドの変数は、次のとおりです。minute、hour、day、month、および weekday の変数には、単一の整数、ワイルドカード（*）、カンマ区切りのリスト、またはダッシュ（-）で区切られた範囲を入力できます。

- *name*：設定レコードの名前。16 文字以内のテキスト文字列を引用符で囲まずに入力します。
- *minutes*：このコマンドを実行する時間（分単位）。0 ～ 59 の整数を入力します。
- *hour*：時間（24 時間単位）。0 ～ 23 の整数を入力します。
- *day*：日付。0 ～ 31 の整数を入力します。
- *month*：月。1 ～ 12 の整数を入力します。

- *weekday*：曜日。1 ～ 7 の整数値を入力します。日曜日は 1 です。
- *command*：実行するコマンド。255 文字以内のテキスト文字列を引用符で囲んで入力します。複数のコマンドはそれぞれセミコロン（;）で区切ります。コマンド文字列に引用符で囲まれた文字が含まれている場合は、一重引用符を使用します。バックスラッシュ（\）が前についていない一重引用符は、コマンド文字列が実行されるときに二重引用符に変換されます。
- *logfile_name*：ログ ファイルの名前を定義するオプションの変数。32 文字以内のテキスト文字列を入力します。

時間変数は、次の値の 1 つまたはいくつかを組み合わせて指定できます。

- 指定した時間変数に 1 つまたは正確な値を定義する 1 つの数値
- 指定した時間変数の有効数値すべてに一致するワイルドカード（*）文字
- 時間変数に複数の値を定義するカンマ区切りの数値リスト（40 文字以内）
- 時間変数に値の範囲を指定するダッシュ（-）文字で区切られた 2 つの数値

たとえば、次のように入力します。

```
(config)# cmd-sched record periodic_shows 30 21 3 6 1 "show
history;show service;show rule;show system-resources"
```

コマンド スケジューラを有効にするには、次のように入力します。

```
(config)# cmd-sched
```

コマンド スケジュールを無効にするには、次のように入力します。

```
(config)# no cmd-sched
```

設定レコードを削除するには、次のように入力します。

```
(config)# no cmd-sched periodic_shows
```

設定されたコマンド スケジューラ レコードの表示

コマンド スケジューラの状態とスケジューリングされている CLI コマンドのレコードに関する情報を表示するには、**show cmd-sched** コマンドを使用します。シンタックスとオプションは、次のとおりです。

- **show cmd-sched**：コマンド スケジューラの状態とスケジューリングされている CLI コマンド レコードをすべてリストアップする。
- **show cmd-sched name** *record_name*：指定した時間にスケジューリングされている CLI コマンド レコードに関する情報をリストアップする。

たとえば、コマンド スケジューラの状態とスケジューリングされている CLI コマンド レコードをすべて表示するには、次のように入力します。

```
(config)# show cmd-sched
```

表 8-1 に、**show cmd-sched** コマンドで表示されるフィールドについて説明します。

表 8-1　show cmd-sched コマンドのフィールド

フィールド	説明
Cmd Scheduler	コマンド スケジューラの状態（有効または無効）および設定されているレコードの数
Sched Rec	設定レコードの名前
Id	レコードの ID
Next exec	レコードが実行される日時
Executions	レコードが実行される回数
MinList	コマンドを実行する時間（分）
HourList	コマンドを実行する時間（時間）
DayList	コマンドを実行する日付
monthList	コマンドを実行する月
WeekdayList	コマンドを実行する曜日。日曜日が 1 です。
Cmd	実行するコマンド。複数のコマンドはそれぞれセミコロン（;）で区切ります。

echo コマンド

スクリプトを実行すると、デフォルトでは各コマンドとその出力が表示されます。**echo** コマンドを使用して、スクリプト実行中の画面表示を制御できます。ほとんどのスクリプトは、役立つ情報を分かりやすく出力することが目的であるため、**no echo** コマンドを使用して **echo** コマンドを無効にすることをお勧めします。**no echo** コマンドは、処理中のコマンドまたはその出力を端末に表示しないようにスクリプト エンジンに指示します。**echo** コマンドを無効にした後に、画面に文字を表示するには **echo** コマンドを明示的に使用する必要があります。

echo は、スクリプト終了時に自動的に有効になります。**echo** コマンドと **no echo** コマンドの詳細については、この章で後述する「!no echo コメント」を参照してください。

（注）　後述するすべての例とその出力では、特に指示のない限り、**echo** コマンドが無効になっていることを想定しています。

コメント行

他のユーザが理解し、保守できるスクリプトを作成するためにも、スクリプトにはコメントを付けることが大切です。コメントは、他のユーザがスクリプトを使用したり、変更する場合、特に重要です。スクリプトにはコメントを、コメント行として簡単に追加できます。スクリプト内の任意の行で、先頭文字として感嘆符（!）を入力すると、その行がコメントになります。

たとえば、次のように入力します。

```
! I want a comment here so that I can tell you that I will
! execute a show variables command
show variables
```

この例は、ユーザ向けのコメントで始まっています（感嘆符に注目してください）。コメント記号の後には任意の文字を使用できますが、行全体がコメントになります。このスクリプトでは、3 行目で **show variables** コマンドを実行します。この行は感嘆符で始まりません。

次の例は有効な文です。

```
! Say hello
echo “Hello”
```

次の例は無効な文です。

```
echo “Hello” ! Say hello
```

!no echo コメント

スクリプトの出力に文字列 no echo を表示せずに **echo** コマンドを無効化するには、スクリプトの先頭行にコメント化した **no echo** コマンドを記述します。コメント化した **no echo** コマンドは、実際にはスクリプトで実行されます（MS-DOS のバッチファイルに詳しいユーザなら、このコマンドが DOS コマンドの **@echo off** と同様だと理解できるでしょう）。たとえば、次のように入力します。

```
! no echo
echo "Hello"
```

出力は次のようになります。

```
Hello
```

たとえば、次のように入力します。

```
! Print Echo
echo "Hello"
```

出力は次のようになります。

```
echo "Hello"Hello
```

このスクリプトはスクリプト エンジンに対して、画面に **echo** コマンドを出力するように指示しているため、このような出力結果になります。スクリプト エンジンが **echo** コマンドとその引数（Hello）を画面に出力し、続いて echo コマンドの出力（Hello）を出力しています。通常、これは望ましい結果ではないので、ほとんどのスクリプトでは先頭行を **!no echo** コマンドで開始するのが一般的です。

変数

CLI では、コマンド、コマンド エイリアス、およびスクリプトを構築するためのユーザ定義変数をサポートします。変数は整数型または文字型で、大文字と小文字は区別されます。変数は、単一の要素または複数要素の配列のいずれかとして存在できます。配列はすべて文字型ですが、その要素は整数式で使用できます。配列の詳細については、この章で後述する「配列」を参照してください。

変数はスクリプト内で作成し、削除できます。スクリプトの終了処理中、スクリプト エンジンはスクリプトが作成した変数をメモリから自動的に削除します。また、スクリプト終了後のセッション環境まで維持されるセッション変数も作成できます。セッション変数をユーザ プロファイルに保存すると、その変数はログイン時に CLI セッション環境で再作成されます。セッション変数をユーザ プロファイルに保存する方法については、第 3 章「ユーザ プロファイルの設定」を参照してください。

変数名には 1 ～ 32 文字を使用できます。値は、通常、英数字を含む文字列を引用符で囲んで指定します。引用符で囲まれた文字列中にスペースを挿入して、配列要素を区切ります。

変数の作成と設定

変数を作成し、変数に値を設定するには、**set** コマンドを使用します。たとえば、次のように入力します。

```
set MyVar "1"
```

このコマンドは、変数 MyVar の値を 1 に設定します。値を指定せずに、変数をメモリに確保することもできます。たとえば、次のように入力します。

```
set MyVar ""
```

これは、変数を NULL（値なし）に設定します。NULL は 0 とは異なります。0 は値です。すべての CLI セッションで使用できるように、変数を設定することができます。たとえば、次のように入力します。

```
set MyVar "1" session
```

session キーワード付きで作成した変数をユーザ プロファイルに保存すれば、その変数は複数の CLI セッションにわたって使用できるようになります。

変数を使用するには、CLI に変数を認識させるために、変数インジケータを指定する必要があります。CLI は処理するテキストの各行で、変数インジケータ $ を検索します。変数インジケータの直後には左波カッコ（{）を*必ず*指定し、続いて変数名を指定します。変数名は、右波カッコ (}) で閉じます。たとえば、**echo** コマンドを使用して画面に変数名を表示する場合は、次のように入力します。

```
set MyVar “CSS11506”
echo “My variable name is: ${MyVar}”
```

次のように出力されます。

```
My Variable name is: CSS11506
```

変数の型

CLI は変数を、整数型または文字型として格納します。CLI は変数の型を、その値によって判別します。値に数字以外の文字が含まれる場合、変数は文字型になります。値が数字だけを含む場合、変数は整数型になります。

引用符で囲まれた "100" などの数値に算術演算を行うことはできますが、"CSS11506" などの文字列には行えません。"CSS11506" が数値でないことが CLI によって自動的に認識されるためです。コマンド文字列の末尾に [*] を付加すると、変数の型が得られます。たとえば、次のように入力します。

```
set MyVar “CSS11506”
echo “My variable type is ${MyVar}[*].”
set Number “100”
echo “My variable type is ${Number}[*].”
```

このスクリプトの出力は次のとおりです。

```
My variable type is char.
My variable type is int.
```

int は整数型（小数部のない数値）、char は文字型（表示可能な任意の ASCII 文字）を意味します。整数型以外の値はすべて文字型と見なされます。CLI は小数点（.）を数字と見なさないため、"3.14" として定義した変数は文字型となります。

変数の削除

メモリから変数を削除するには、**no set** コマンドを使用します。

たとえば、次のように入力します。

```
no set MyVar
```

変数 MyVar が存在すれば、このコマンドはメモリから MyVar を削除します。指定した変数が存在しない場合、無効なアクションを通知するエラー メッセージが CLI に表示されます。

（注）　セッション変数を恒久的に削除するためには、その変数をユーザ プロファイルからも削除する必要があります。

整数型変数の変更

ここでは、次の内容について説明します。

- no set コマンドと set コマンド
- 算術演算子
- 増分演算子と減分演算子

no set コマンドと set コマンド

変数を変更するには、**no set** コマンドを使用してメモリから変数を削除し、続いて **set** コマンドで同じ名前の変数に別の値を割り当てます。**set** コマンドで作成した変数を **no set** で削除せず、同じ変数に繰り返し **set** を実行することもできます。

例 1

```
set MyVar "1"
no set MyVar
set MyVar "2"
```

例 2

```
set MyVar "1"
set MyVar "2"
```

例 3

```
set MyVar1"1"
set Myvar2 "2"
set MyVar1 "${MyVar2}"
```

（注）　**set** コマンドと **no set** コマンドは文字変数にも使用できます。

算術演算子

算術演算子（-、+、/、*、MOD）を使用して変数の値を変更するには、**modify** コマンドを使用します。たとえば、次のように入力します。

```
set MyVar "100"
modify MyVar "+" "2"
echo "Variable value is ${MyVar}."
modify MyVar "-" "12"
echo "Variable value now is ${MyVar}."
modify MyVar "*" "6"
echo "Variable value now is ${MyVar}."
modify MyVar "/" "6"
echo "Variable value now is ${MyVar}."
modify MyVar "MOD" "10"
echo "Variable modulus value now is ${MyVar}"
```

次のように出力されます。

```
Variable value is 102.
Variable value now is 90.
Variable value now is 540.
Variable value now is 90.
Variable modulus value now is 0.
```

単純な算術演算の場合、**modify** コマンドは引用符で囲まれた演算子（"/"、"*"、"+"、"-"、"MOD" など）と、同じく引用符で囲まれた新しい値（オペランド）を取ります。オペランドは定数（"5"、"10" など）だけでなく、他の変数（"${Var1}"、

"${Var2}" など）も使用できます。剰余演算子 "MOD" を使用すると、指定したオペランド（上記の例では "10"）で変数値を割った余りが、変数に割り当てられます。

次の項では、**modify** コマンドを使用して変数に実行できるその他の演算について説明します。**modify** コマンドの詳細については、『*Cisco Content Services Switch Command Reference*』を参照してください。

増分演算子と減分演算子

CSS のスクリプト言語には、変数値の増分と減分のために 2 つの演算子があります。増分演算子 "++" は変数値に 1 を加え、減分演算子 "--" は変数値から 1 を引きます。これらの演算子は **modify** コマンドで使用します。

たとえば、次のように入力します。

```
set MyVar "1"
echo "Variable is set to ${MyVar}."
modify MyVar "++"
echo "Variable is set to ${MyVar}."
modify MyVar "--"
echo "Variable is set to ${MyVar}."
```

次のように出力されます。

```
Variable is set to 1.
Variable is set to 2.
Variable is set to 1.
```

これら 2 つの演算子を使えば、オペランドを入力せずに加算や減算を行うことができます。したがって、次のように置き換えることができます。

```
modify MyVar "+" 1
```

上記を次の式で置き換えます。

```
modify MyVar "++"
```

（注）　増分演算子と減分演算子はどちらも、整数型の変数専用です。これらの演算子を文字型の変数値（"CSS11506" など）に使用するとエラーになります。

論理 / 関係演算子と分岐コマンド

構造化したコマンド ブロックを構築するには、**if** および **while** 分岐コマンドを使用します。**if** コマンドは、先行するコマンドの結果に基づいてスクリプトの処理を分岐させます。**while** コマンドはループ メカニズムです。これらのコマンドは、繰り返し処理の少ない効率的なスクリプトの作成に役立ちます。

これらの分岐コマンドは、どちらも **endbranch** コマンドと組み合わせて使用します。endbranch は、コマンドの論理ブロックの終了を CLI に通知します。対応する **endbranch** コマンドを持たない分岐は、すべてスクリプト論理エラーになり、スクリプト シンタックス エラーが生成される場合もあります。スクリプト エラーの詳細については、この章で後述する「シンタックス エラーとスクリプトの終了」を参照してください。

（注）　分岐コマンドは最大 32 レベルまでネストできます。

ブール論理演算子と関係演算子

次の演算子は **if**、**while**、または **modify** コマンドと共に使用できます。

分岐コマンドで使用できるブール論理演算子は、次のとおりです。

- **AND**：論理積
- **OR**：論理和

分岐コマンドで使用できる関係演算子は、次のとおりです。

- **GT**：大なり
- **LT**：小なり
- **==**：等しい
- **NEQ**：等しくない
- **LTEQ**：小なりまたは等しい
- **LTEQ**：大なりまたは等しい

if 分岐コマンド

先行するコマンドの結果に基づいてスクリプトの処理を分岐するには、**if** コマンドを使用します。先行するコマンドの結果が **if** 文の条件式を満たす場合、スクリプト エンジンは、**if** と **endbranch** 間のすべての行を実行します。条件式が満たされない場合には、スクリプト エンジンは次の **endbranch** 文までの間に記述されている全コマンドをスキップし、endbranch の直後の行から実行を継続します。

```
set MyVar "1"
if MyVar "==" "1"
    echo "My variable is equal to ${MyVar}!!!"
endbranch
if MyVar "NEQ" "1"
    echo "My variable is not equal to 1 (oh well)."
endbranch
```

上記のスクリプト例は、変数 MyVar の値が "1" と等しいかどうかをチェックしています。値が等しければ、**if** と **endbranch** の間に記述された **echo** コマンドが実行されます。変数 MyVar の先頭に変数インジケータ（$）が付けられていないことに着目してください。これは、**if** コマンドが直後に定数または変数名を必要とするためです。

ただし、**if** コマンドが配列の要素を参照する場合は例外です。その場合には、変数インジケータ（$）と波かっこ（{ }）を含む、通常の変数シンタックスを使用する必要があります。配列については、この章で後述する「配列」を参照してください。

たとえば、次の論理ブロックは有効です。

```
if 12 "==" "${MyVar}"
    echo "We made it!"
endbranch
```

一方、次の論理ブロックは無効になります。

```
if "12" "==" "${MyVar}"
    echo "We made it!"
endbranch
```

if コマンドは直後に定数または変数名（変数インジケータなし）を要求するため、文字列 "12" はこの条件を満たしません。

また、変数値が NULL かどうかをチェックすることもできます。たとえば、次のように入力します。

```
if MyVar
    echo "MyVar is equal to ${MyVar}"
endbranch
```

while 分岐コマンド

関連する式の結果に基づいて、繰り返し同じコマンドを実行するには、**while** 分岐コマンドを使用します。式の結果が真（1 以上）であれば、スクリプト エンジンは分岐内のコマンドを実行します。結果が偽（0）の場合には、スクリプトはそのままループを抜け、次の **endbranch** コマンドの直後の行から実行を継続します。たとえば、次のように入力します。

```
set Counter "0"
while Counter "NEQ" "5"
    echo "Counter is set to ${Counter}."
    modify Counter "++"
endbranch
echo "We're done!"
```

この論理ブロックの出力は、次のとおりです。

```
Counter is set to 0.
Counter is set to 1.
Counter is set to 2.
Counter is set to 3.
Counter is set to 4.
We're done!
```

スクリプトの制御は、**endbranch** に達するたびにループの先頭に戻って式を評価し、式が*満たされなくなる*までループの実行を繰り返します。スクリプトの制御がループを 5 巡するまでの間、変数 Counter の値は 0、1、2、3、4 と、それぞれの巡回で 1 ずつ増加します。5 回目のループが終了した時点で Counter の値は 5 に等しくなり、"While Counter is not equal to 5" という式を満たしません。したがって、式の結果は偽になり、そこでループが終了します。

if コマンドの場合と同様、**endbranch** コマンドは **while** コマンドの論理ブロックを終了します。

特殊変数

CLI には、スクリプト内で使用して、スクリプトの機能をいっそう強化できる一連の定義済み変数があります。これらの定義済み変数（特殊変数）のうち、一部はスクリプトの動作を変更する変数であり、残りは情報提供だけを目的とした変数です。特殊変数はいずれも、ユーザ自身が明示的にメモリからクリアする必要はありません。ただし、CONTINUE_ON_ERROR と EXIT_MSG については、これらを使用するコマンド ブロックの実行完了後に、明示的にクリアすることが推奨されます。

情報変数

情報変数の内容は次のとおりです。

- **LINE**：回線名（pty1、console など）
- **MODE**：コマンドの現在のモード（configure、boot、service など）
- **USER**：現在ログインしているユーザ（admin、bob、janet など）
- **ARGS**：CLI からスクリプトに渡される引数のリスト。この章で後述する「コマンド行引数」を参照してください。
- **UGREP**：**grep -u** コマンドから戻された文字列
- **CHECK_STARTUP_ERRORS**：ユーザのログイン時に起動エラーが生成されたかどうかを示すセッション変数

CONTINUE_ON_ERROR 変数

CONTINUE_ON_ERROR は、対話型 CLI セッションで実行しているスクリプトが、コマンド エラーを検出したときにどのように動作するかを制御する変数です。デフォルトでは、スクリプトはエラーを検出すると終了します。たとえば、**echo** コマンドを使用してスクリプトに情報を出力させる場合、コマンドのスペルが間違っているとシンタックス エラーでスクリプトが終了し、エラーが検出された行が表示されます。たとえば、次のように入力します。

```
! Spell echo incorrectly
eco “This will not print”
```

次のように出力されます。

```
Error in script playback line:2
>>>eco "Hello"
     ^
%% Invalid input detected at '^' marker.
Script Playback cancelled.
```

このスクリプトはコマンドのスペルミスを含んでいるため、エラーの原因を示すメッセージを出力して終了します。

ただし、エラーが発生してもスクリプトの実行を継続する方が望ましい場合もあります。そのような場合に CONTINUE_ON_ERROR 変数を設定すれば、デフォルトの動作を無効にできます。この変数を任意の値に設定すると、シンタックス エラーやその他のエラーを検出した後も、スクリプトは実行を継続するようになります。

（注）　この変数を設定すると CLI はシンタックス エラーを無視するようになるため、使用にあたっては十分に注意してください。スクリプト内のコマンド エラーが予想される部分では、設定したらその後で必ず設定をクリアしてください。

そのようなスクリプトの一例を次に示します。

```
set CONTINUE_ON_ERROR "1"
! Spell echo incorrectly
eco "This will not print"
echo "This will print"
no set CONTINUE_ON_ERROR
```

次のように出力されます。

```
This will print
```

この例では、スクリプトは "Script Playback cancelled" のメッセージを表示しないで終了します。これは、CONTINUE_ON ERROR 変数を設定しているためです。CONTINUE_ON_ERROR 変数に **no set** コマンドを実行する処理は、ほとんどの場合で欠かせません。特定のエラーの発生後もスクリプトの実行を継続すべき場合には、この変数をいったん設定し、目的が果たされた時点で設定をクリアする必

要があります。この変数に **no set** コマンドを実行しないと、スクリプトにその他のシンタックス エラーが含まれる場合でも、スクリプトは途中で終了しません。なお、この変数は 0 に設定しても無効になりません。この変数の機能を無効化するには、設定を明示的にクリアする必要があります。

STATUS 変数

STATUS 変数を使用すれば、直前に実行された CLI コマンドの終了ステータスを得ることができます。**grep** 以外のほとんどのコマンドでは、終了ステータス 0 が正常終了、0 以外がエラーを意味します。STATUS 変数には、コマンドの実行が完了するたびに、CLI によって自動的に値が設定されます。

（注）　**grep** コマンドの場合、STATUS 変数には検索条件を満たした行の数が設定されます。**grep** コマンドの詳細については、この章で後述する「grep コマンド」を参照してください。

通常は、コマンドが正常に実行されないとスクリプトが終了するため、STATUS 変数の値をチェックする必要はありません。ただし、CONTINUE_ON_ERROR 変数を設定している場合は、STATUS 変数を使用してコマンドの結果をチェックできます。

たとえば、次のように入力します。

```
set CONTINUE_ON_ERROR "1"
eco "Hello world"
if STATUS "NEQ" "0"
    echo "Failure to execute command correctly"
endbranch
```

上記の例では、STATUS 変数は 0 以外の値に設定されます。実際に設定される値は、発生したエラーの種類によって異なります。このスクリプトでは、シンタックス エラーの有無だけがチェックされ、実行中のコマンドが失敗したことが通知されます。この例はあくまで STATUS 変数の典型的な使用例に過ぎず、実際に CONTINUE_ON_ERROR 変数を使用するスクリプトでは、さらに工夫を加える必要があります。ほとんどの場合、具体的なシンタックス エラーの内容を知ることが、より重要になります。

（注）　スクリプトを作成する際には、コマンドが実行されるたびに STATUS 変数の値が変わることに注意してください。いったん設定された STATUS 変数の値を、いくつかのコマンドが実行された後、同じスクリプト内で使用するには、STATUS の値を別の変数に割り当てておく必要があります。

次の例では、発生する可能性が最も高いのはシンタックス エラーではなく、開始されるコマンドに関するエラーです。この例でも、STATUS 変数の値が 0 以外であればメッセージ（Failure to connect to remote host）が表示されます。ただし、代替処理を講じるためにも、エラーの有無を知ることが重要という点で、前述の例とは異なります。

```
set CONTINUE_ON_ERROR "1"
socket connect host 1.1.1.1 port 9
if STATUS "NEQ" "0"
    echo "Failure to connect to remote host"
endbranch
no set CONTINUE_ON_ERROR
```

EXIT_MSG 変数

EXIT_MSG 変数を使用すれば、スクリプトの終了時にカスタム メッセージを表示するように、CLI に指示を与えることができます。この変数には通常、スクリプト終了時に表示する文字列を設定します。この変数は、発生する可能性があるエラーに備えて設定し、スクリプトが正常終了する場合には、その直前に **no set** コマンドで設定をクリアします。この変数を使用すると、エラー発生時にただちに終了するという CLI の機能を利用できると同時に、エラー メッセージを自由にカスタマイズできます。たとえば、次のように入力します。

```
set EXIT_MSG "Failure to connect to host"
socket connect host 1.1.1.1 port 9
no set EXIT_MSG
```

この例は、**socket connect** コマンドが STATUS 変数に 0 以外の値を戻したときに表示されるカスタム エラー メッセージ（Failure to connect to host）の作成方法を示しています。このエラーが発生した場合（CONTINUE_ON_ERROR 変数が設定されている場合を除く）、スクリプトは自動的に終了し、EXIT_MSG の文字列が CLI によって画面に出力されます。

socket connect コマンドが成功した場合は、スクリプト内で直後に位置するコマンドが実行されます。この例では、**no set** EXIT_MSG コマンドが実行されます。これにより、スクリプトは終了メッセージを画面に表示しないで正常終了します。エラーが発生していないため、終了メッセージは不要です。

SOCKET 変数

SOCKET 変数には、ホストに対応した接続 ID が格納されます。リモート ホストに接続すると、SOCKET 変数が設定され、この変数を参照してメッセージを送受信できるようになります。**socket** コマンドを使用すると、SOCKET 変数が自動的に設定されます（この章で後述する「socket コマンド」を参照してください）。**socket** コマンドで複数の接続を作成する場合は、SOCKET 変数の値を他の変数に順次割り当ててください。この割り当てを行わないと SOCKET 変数は上書きされ、直前の接続の ID は失われます。

たとえば、次のように入力します。

```
set EXIT_MSG "Failure to connect to host"
socket connect host 1.1.1.1 port 80
no set EXIT_MSG
set EXIT_MSG "Send: Failure"
socket send ${SOCKET} "GET /index.html\n\n"
no set EXIT_MSG
! Save current socket ID
set OLD_SOCKET "${SOCKET}"
! The new socket connect command will overwrite the old
! ${SOCKET} variable
set EXIT_MSG "Failure to connect to host"
socket connect host 1.1.1.1 port 80
no set EXIT_MSG
set EXIT_MSG "Send: Failure"
socket send ${SOCKET} "GET /index.html\n\n"
no set EXIT_MSG
set EXIT_MSG "Waitfor: Failed"
socket waitfor ${OLD_SOCKET} "200 OK"
socket waitfor ${SOCKET} "200 OK"
! Finished, cleanup
no set EXIT_MSG
socket disconnect ${OLD_SOCKET}
socket disconnect ${SOCKET}
```

show variable コマンド

show variable コマンドを使用すると、CSS ソフトウェア環境で現在設定されているすべての変数を表示できます。

CLI は、次に挙げる特殊変数を使用することによってセッションの動作を制御し、CLI コマンドとユーザ間の対話性を向上させます。

- USER 変数。ログイン時に、CLI セッションを開始しているユーザ名が自動的に設定されます。
- LINE 変数。ログイン時に、ユーザが接続している回線に自動的に設定されます。
- MODE 変数。ユーザによる CLI のモード階層間の移動に応じて、自動的に現在のモードに設定されます。
- STATUS 変数。直前に実行した CLI コマンドの終了ステータスを返すように自動的に設定されます。grep 以外のほとんどのコマンドでは、終了ステータス 0 が正常終了、0 以外がエラーを意味します。
- CHECK_STARTUP_ERRORS 変数。プロファイル スクリプト内で設定されている場合、ログイン時の起動エラーの有無を示します。startup-errors ファイルがログ ディレクトリ内に見つかると、起動エラーが発生したことを示すメッセージ（***Startup Errors occurred on boot.***）が表示されます。
- CONTINUE_ON_ERROR 変数。対話型 CLI セッションで実行しているスクリプトが、コマンド エラーを検出したときにどのように動作するかを決定します。この変数をスクリプト内で、**set** コマンドを使用して設定すると、エラーが検出されてもスクリプトの実行は継続されます。この変数をスクリプト内で設定しないと、そのスクリプトはエラーの発生により終了します。

  この変数を使用する際は、注意が必要です。この変数を設定すると、シンタックス エラーは無視されるようになります。そのため、スクリプト内のコマンド エラーが予想される部分では設定したらその後で、**no set** コマンドを使って必ず設定をクリアしてください。

たとえば、次のように入力します。

```
show variable
```

次のように出力されます。

```
$MODE = super
$LINE = console
$CHECK_STARTUP_ERRORS = 1 *Session
$UGREP =   Weight:  1           Load:  255
$SOCKET = -1
$USER = admin
$STATUS = 0
```

この例では、いくつかの変数がすでに環境内で定義されている点に着目してください。パラメータとして特定の変数名を指定して **show variable** コマンドを実行し、その変数の値を表示することもできます。

たとえば、次のように入力します。

```
show variable LINE
```

次のように出力されます。

```
$LINE = console
```

配列

変数は、そのメモリ空間にサブ値（要素）を保持できます。そのような変数は通常、変数配列（または単に配列）と呼ばれます。配列には、数値、文字列、またはその両方を格納できます。配列を作成するには、**set** コマンドで変数を作成し、スペースで区切った一連の配列要素を含む文字列を割り当てます。たとえば、次のように入力します。

```
set WeekDays "Sun Mon Tues Wed Thurs Fri Sat"
```

配列の値は、他の変数と同じように表示できます。たとえば、次のように入力します。

```
echo "Days of the week: ${WeekDays}."
```

次のように出力されます。

```
Days of the week: Sun Mon Tues Wed Thurs Fri Sat.
```

ただし、曜日を個別に表示するには、配列内の特定の要素を参照する必要があります。たとえば、次のように入力します。

```
echo "The first day of the week is ${WeekDays}[1]."
echo "The last day of the week is ${WeekDays}[7]."
```

次のように出力されます。

```
The first day of the week is Sun.
The last day of the week is Sat.
```

要素番号を角カッコ（[ ]）で囲んで指定して、使用する要素を CLI に指示することによって、特定の配列要素を参照しています。角カッコは変数名（変数インジケータと波カッコを含む）の直後に付加します。

（注）　CSS スクリプト言語では、要素番号は 0 ではなく 1 から始ま ります。スクリプト言語やプログラミング言語によっては、配列のインデックスを 0 からカウントする場合もありますが、このスクリプト システムは異なります。

配列の境界を超える参照を試みると、シンタックス エラーが発生します。そのような例を次に示します。

```
echo "The last day of the week is ${WeekDays}[8]"
%% Error variable syntax
```

この例では、配列 WeekDays 内に存在しない 8 番目の要素の参照を試みた結果、エラーが発生しています。

要素番号

配列内の要素数を確認するには、要素番号の代わりにシャープ記号（#）を使用します。たとえば、1 週間の日数を知りたい場合は、次のように指定します。

```
set WeekDays "Sun Mon Tues Wed Thurs Fri Sat"
echo "There are ${WeekDays}[#] days in a week."
```

次のように出力されます。

```
There are 7 days in a week.
```

この方法を使用して、**while** コマンドの実行回数を判断できます。次に例を示します。

```
set WeekDays "Sun Mon Tues Wed Thurs Fri Sat"
set Counter "1"
while Counter "LTEQ" "${WeekDays}[#]"
    echo "Counter is set to ${Counter}."
    modify Counter "++"
endbranch
```

次のように出力されます。

```
Counter is set to 1.
Counter is set to 2.
Counter is set to 3.
Counter is set to 4.
Counter is set to 5.
Counter is set to 6.
Counter is set to 7.
```

var-shift コマンドによる配列要素の取得

1 週間のすべての曜日を画面出力する場合など、配列内のすべての要素の出力が必要になることもあります。各要素の値をスクリプト内に直接記述して出力する方法も考えられますが、実際には現実的な方法ではなかったり、あるいは不可能な場合さえあります。1 週間の曜日なら 7 つの要素があるのは明白ですが、スクリプトを実行するまで配列内の要素数が分からない場合もあるためです。

var-shift コマンドを使用すると、配列から要素を一度に 1 つずつ取り出すことができます。たとえば、次のように入力します。

```
set WeekDays "Sun Mon Tues Wed Thurs Fri Sat"
while ${WeekDays}[#] "GT" "0"
    ! Push the 1st element out of the array and shift all
    ! elements up one position.
    echo "Day: ${WeekDays}"
    var-shift WeekDays
endbranch
```

この論理ブロックの出力は、次のとおりです。

```
Day: Sun
Day: Mon
Day: Tues
Day: Wed
Day: Thurs
Day: Fri
Day: Sat
```

変数をインデックスとして使用して、配列内の特定の要素を取得することはできません。したがって、次の文は無効となります。

```
set Pet "Dog Cat"
set Index "1"
echo "First Pet is: ${Pet}[${Index}]"
modify Index "+" 1
echo "Second Pet is: ${Pet}[${Index}]"
```

次のように記述した場合は有効になります。

```
set Pet "Dog Cat"
echo "First Pet is: ${Pet}[1]"
var-shift Pet
echo "Second Pet is: {$Pet}[1]"
```

2 番目の例では使用している変数が 1 つ少なく（Index は不要）、参照している要素も先頭要素（${Pet}[1]）だけです。

var-shift コマンドで要素を取り出すと、その要素は配列から削除されます。上記の例では、実行後に変数 Pet に格納されている要素は Cat だけです。この問題を回避するには、元の変数の内容を一時変数にコピーします。

たとえば、次のように入力します。

```
set Pet "Dog Cat"
set Temp "${Pet}"
echo "First Pet is: ${Temp}[1]"
var-shift Temp
echo "Second Pet is: {$Temp}[1]"
no set Temp
```

変数 Temp を使用すると、元の変数はそのまま残ります。Temp は不要になりしだい、**no set** コマンドでメモリから削除します。

ユーザ入力の取り込み

ユーザ入力を取り込んで変数に格納するには、**input** コマンドを使用します。このコマンドを使用すると、ユーザによる設定ファイルの編集を支援したり、あらかじめ定義した方法で CSS をセットアップする際に役立つスクリプトを作成できます。たとえば、次のように入力します。

```
! Ask the user for his/her full name
echo "What is your full name?"
input FULL_NAME
echo "Hello ${FULL_NAME}!"
```

この例では、**input** コマンドに引数として変数 FULL_NAME を指定しています。この FULL_NAME には、**input** コマンドより前の部分で、何も設定を行っていません。このコマンドを実行すると変数 FULL_NAME が作成され、ユーザから入力された値が格納されます。なお、ユーザ入力の末尾は、常に改行文字になることに留意してください。

ユーザは任意の英数字を入力できます。ユーザが Enter キーだけを押して、文字を入力しなかった場合、このスクリプトは値 NULL の変数を作成します。したがって、値が NULL かどうかを調べれば、有効な入力の有無をチェックできます。次の例では、ユーザが「y」を入力するまで同じ質問が繰り返し表示されます。

```
echo -n "\please enter the character 'y' to exit."
input DATA
while DATA "NEQ" "y"
    echo -n "Please enter the character 'y' to exit: "
    input DATA
    echo "\n"
endbranch
```

この例では、**echo** コマンドにパラメータ *-n* を指定している点に着目してください。このパラメータは、出力メッセージの末尾の改行を抑制します。したがって、ユーザが入力したデータは、**echo** コマンド出力に続いて、同じ行に表示されます。**echo** コマンドに渡す引用符で囲んだ文字列には、改行文字（\n）を埋め込むことができます。この改行文字は C 言語の文字列の場合と同じで、出力を読みやすく整えることができます。

コマンド行引数

CLI では、**script play** コマンドを使用することによって、引用符で囲んだ文字列をコマンド行引数としてスクリプトに渡すことができます（「スクリプトの実行」を参照）。コマンド行からスクリプトに渡された引数は、予約済みの特殊変数 ARGS に格納されます。

次の文は、スクリプトに渡されたすべてのコマンド行引数を表示します。

```
echo "You passed the arguments: ${ARGS}"
```

それぞれの引数を個別に参照するには、ARGS 変数を配列として使用します。コマンド行から渡された各引数は、互いにスペースで区切られています。次に例を示します。

```
echo "Your first argument passed is: ${ARGS}[1]"
```

次のスクリプト（NameScript）は、ユーザの名前と姓を画面に出力します。このスクリプトを使用するには、ユーザの名前と姓をこの順序でスペースを挟んで連結し、全体を引用符で囲んだ文字列を、このスクリプトに渡す必要があります。スクリプト（NameScript）の内容は次のとおりです。

```
!no echo
if ${ARGS}[#] "NEQ" "2"
    echo "Usage: NameScript \'First_Name Last_Name\'"
    exit script 1
endbranch
echo "First Name: ${ARGS}[1]"
echo "Last Name: ${ARGS}[2]"
exit script 0
```

このスクリプトは最初に、ユーザが正しく引数を渡したかどうかをチェックしています。ユーザが 2 つの引数を渡さなかった場合、スクリプトは画面に使用方法を出力して終了します。ユーザが 2 つの引数を渡した場合は、最初の引数がユーザの名前、2 番目の引数が姓と見なされます。最後に、ユーザの名前と姓を画面に出力します。

この NameScript を実行するには、次のように入力します。

```
script play NameScript "John Doe"
```

次のように出力されます。

```
First Name: John
Last Name: Doe
```

関数

関数を使用すると、スクリプト内にサブルーチンやモジュールを作成できます。作成した関数は、スクリプトで必要になったときに呼び出すことができます。

読みやすさと単純化の両面からスクリプトをモジュール化するには、**function** コマンドを使用します。このコマンドでは、スクリプト関数の作成と呼び出しの両方を行うことができます。たとえば、次のように入力します。

```
echo “Calling the PrintName function”
function PrintName call
echo “End”
! Function PrintName: Prints the name John Doe
function PrintName begin
echo “My Name is John Doe”
function PrintName end
```

次のように出力されます。

```
Calling the PrintName function
My Name is John Doe
End
```

function PrintName begin と **function PrintName end** コマンドの間に記述されているコマンドは、スクリプトの最後の **echo** 文より先に実行されます。また、このスクリプトは、有効な最終行に制御が達すると終了し、そのまま関数定義に制御が移動することはありません。

関数への引数の引き渡し

関数には、いくつかの引数を渡すことができます （これは、スクリプトにコマンド行引数を渡す場合と似ています）。また、スクリプトで関数を呼び出すときにこれらの引数を使用することもできます。次に例を示します。

```
echo “Calling the PrintName function”
function PrintName call “John Doe”
echo “End”
! Function PrintName: Prints the name John Doe
function PrintName begin
echo “My Name is ${ARGS}”
function PrintName end
```

次のように出力されます。

```
Calling the PrintName function
My Name is John Doe
End
```

このスクリプトでは、関数内で変数 ARGS を使用して、渡された引数を保持しています。コマンド行引数を **script play** コマンドで渡すように、関数の引数は **function call** コマンドで渡します。

コマンド行引数を **script play** コマンドでスクリプトに渡し、そのスクリプト内で関数に引数を渡す場合でも、コマンド行引数を格納する変数 ARGS の値は、関数から制御が戻るまで維持されます。

たとえば、**script play** コマンドで次のスクリプトに 2 つの引数 "Billy Bob" を渡す場合について考えてみましょう。このスクリプト内では、関数 PrintName に別の引数が渡されています。

```
echo “I was passed the arguments ${ARGS}”
function PrintName call “John Doe”
echo “The original arguments are ${ARGS}”
! Function PrintName: Prints the name John Doe
function PrintName begin
echo “My Name is ${ARGS}”
function PrintName end
```

次のように出力されます。

```
I was passed the arguments Billy Bob
My Name is John Doe
The original arguments are Billy Bob
```

このスクリプトでは変数 ARGS が 2 箇所で使用されていますが、関数はメインスクリプトとは別に専用の ARGS を持つため、これら 2 つの ARGS にはそれぞれ異なる値が格納されています。

関数を利用してスクリプトをモジュール化すれば、記述内容がいっそうわかりやすくなり、保守も容易になります。

SCRIPT_PLAY 関数

スクリプト内から他のスクリプトを呼び出すには、**SCRIPT_PLAY** 関数を使用します。このコマンドのシンタックスは次のとおりです。

function SCRIPT_PLAY call *"ScriptName arg1 arg2..."*

SCRIPT_PLAY 関数を使用する場合には、**exit script** コマンドの扱いに注意する必要があります。スクリプトの終了方法の詳細については、この章で後述する「スクリプト内での他のスクリプトの終了」を参照してください。

ビット演算子

CSS のスクリプト言語は、ビット操作用の 2 種類の演算をサポートしています。これらの演算の対象は数値だけです。次のビット演算子があります。

- **BAND**：ビット論理積
- **BOR**：ビット論理和

数値変数のビットを操作するには、**modify** コマンドを使用します。たとえば、13 という数値の 2 番目と 4 番目のビットがオン（1）かどうかを確認するには、次の方法でビット論理積を求めます。

```
        00001101 (13)
AND     00001010 (10)
```

結果は 00001000 (8) になります。

13 と 10 は 4 番目のビットが共通であるため、このビットだけを立てた値がビット論理積になります。この演算は、スクリプトでは次のように記述できます。

```
set VALUE "13"
modify VALUE "BAND" "10"
echo "Value is ${VALUE}"
```

次のように出力されます。

```
Value is 8
```

（注）　このスクリプトを実行すると、変数 VALUE の元の値は上書きされます。BOR 演算子の使用方法もほぼ同様です。AND および OR 演算の仕組みは本書の対象外であり、ここでは説明しません。

シンタックス エラーとスクリプトの終了

CSS のスクリプト言語を使用すれば、複雑なスクリプトを作成できます。スクリプトの開発中には、いくつかのシンタックス エラーが発生することがあります。シンタックス エラーでスクリプトが終了すると、シンタックス エラーが発生した行の番号と記述内容が CLI に表示されます。

exit script コマンドを使用すれば、終了コード（0 または 0 以外）を指定してスクリプトを終了させることもできます。この方法では、スクリプトが失敗した理由を的確に判断することが可能です。さらに、得られた情報を意思決定プロセスに反映すれば、エラー状況に効果的に対処できます。

シンタックス エラー

スクリプトの実行中に綴りの誤ったコマンドや不明なコマンドが検出されたり、コマンドの実行に失敗したときには、画面にシンタックス エラー メッセージが表示されます。このエラー メッセージには、スクリプト内でエラーが発生した行の番号と記述内容が示されています。また、STATUS 変数には、CSS ソフトウェアによって適切なエラー コード（0 以外の値）が格納されます。

次にシンタックス エラーの一例を示します。

```
Error in script playback in line: 1
>>>eco “Hello”
    ^
%% Invalid input detected at '^' marker.
Script Playback cancelled.
```

スクリプト内で実行されたコマンドが 0 以外の STATUS コードを返した場合も、シンタックス エラーになります。たとえば、スクリプト内で **socket connect** コマンドが実行されたときに、ホストによって接続が拒否されるか、ホスト名の解決エラーが発生すると、このスクリプトはシンタックス エラーで終了します。

たとえば、次のように入力します。

```
!no echo
socket connect host 192.168.1.1 port 84 tcp
socket disconnect ${SOCKET}
```

このスクリプトは、2 つのコマンドがどちらも失敗しなければ正常に動作しますが、ホスト 192.168.1.1 が存在しない場合には 次のエラーを生成します。

```
Error in script playback line:2
>>>socket connect host 192.168.1.1 port 84 tcp
CSS11506#
Script Playback cancelled.
```

2 行目にエラーがあるため、スクリプトは失敗しました。コマンドの綴りとシンタックスには問題がありませんが、ホストに接続できなかったために 0 以外のエラー コードが戻されました。3 行目のコマンド（**socket disconnect**）は、最初のコマンドが失敗したため実行されません。

CLI では、このようなエラーもシンタックス エラーと見なされます。エラーの原因を見つけるには、原因と思われるコマンドを CLI で直接実行します。

たとえば、次のように入力します。

```
socket connect host 192.168.1.1 port 84 tcp
%% Failed to connect to remote host
```

コマンドが失敗した理由が画面に表示されます。

（注）　スクリプトに綴りのミスが見つからない場合、CLI でコマンドをテストすることをお勧めします。

スクリプトの終了コード

スクリプトは、終了時に終了コードを返します。終了コードはソフトウェアによって変数 STATUS に格納され、スクリプト終了後に確認できます。スクリプトの終了コードの値は 0（成功）または 0 以外（失敗）です。

スクリプトが正常終了した場合の終了コードを明示的に指定するには、**exit script** コマンドに 0（デフォルト）を付けて使用します。このコマンドでは整数値はオプションです。

たとえば、次のように入力します。

```
! Exit Cleanly
exit script 0
```

スクリプトが正常に実行されなかったときに、終了コードを出力させる方が適切な場合もあります。たとえば、ユーザに 1 つ以上のコマンド行引数の入力を求めるスクリプトなどです。ユーザが指定した引数の内容をシンタックスでチェックすることはできませんが、指定した引数の数はチェックできます。次に一例を示します。

```
if ${ARGS}[#] "NEQ" "2"
    echo "Usage: PingScript \'HostName\'"
    exit script 1
endbranch
```

このスクリプトは、コマンド行引数の数が 2 以外の場合、ステータス コード 1（失敗）で終了します。この時点で変数 STATUS をチェックすると、値は 1 に設定されています。

（注）　CLI のすべてのコマンドは、終了時に変数 STATUS に終了コードを書き込みます。STATUS の値を使用する場合は、他の変数に値をコピーするか、ただちに使用する必要があります。

出力する場合は、次のように記述します。

```
script play PingScript
echo "Status: ${STATUS}"
echo "Status: ${STATUS}"
```

次のように出力されます。

```
Usage: PingScript "HostName"
Status: 1
Status: 0
```

このスクリプトには 1 を指定した **exit script** コマンドが含まれているため、最初の **echo** コマンドは STATUS として 1 を返し、スクリプトが失敗したことを示します。2 番目の **echo** コマンドは、最初の **echo** コマンドが正常に実行されたため、STATUS として 0 を返します。

スクリプト内での他のスクリプトの終了

SCRIPT_PLAY 関数を使用すれば、スクリプト内から他のスクリプトを実行できます。スクリプトの実行の詳細については、この章で前述した「SCRIPT_PLAY 関数」を参照してください。この場合、2 番目のスクリプト内の **exit script** コマンドの扱いには、十分に注意する必要があります。

スクリプト A でスクリプト B を呼び出し、スクリプト B で **exit script** コマンドを発行すると、両方のスクリプトが終了してしまいます。したがって、他のスクリプトを呼び出すスクリプトでは、2 番目のスクリプトの **exit script** コマンドを削除するか、あるいはその他の措置を行って、この動作に対処することが重要になります。

grep コマンド

指定したデータを検索し、検索結果の最終行を変数 UGREP に格納するには、**grep** コマンドを -u オプション付きで使用します。たとえば、サービス S1 の **show service** コマンドの実行結果に含まれる Keepalive フィールドを検索するには、次のようなスクリプトを作成します。

```
!no echo
show service S1 | grep -u "Keepalive"
echo "The line is: ${UGREP}"
```

次のように出力されます。

```
The line is: Keepalive: (SCRIPT a-kal-pop3 10 3 5)
```

show service の出力画面には Keepalive フィールドが含まれているため、行全体が変数 UGREP に格納されます。さらに、変数 UGREP を配列として扱い、スペースで区切った各要素を抽出することもできます。たとえば、この配列の最初の要素を抽出するには、次のように記述します。

```
!no echo
show service S1 | grep -u "Keepalive"
echo "The first element in the line is: ${UGREP}[1]"
```

次のように出力されます。

```
The first element in the line is: Keepalive:
```

検索結果の行番号の指定

grep コマンドの検索結果が、複数の行になる場合もあります。その場合には、**grep** -u*[n]*（*[n]* はオプションの行番号）の形式で実行すれば、変数 UGREP に格納する行番号を指定できます。デフォルトでは、UGREP には検索結果の最終行が格納されます。サービス S1 の **show service** コマンドの実行結果に **grep** コマンドを実行し、**show service** の出力画面に含まれるすべてのコロン（:）を検索するとします。たとえば、次のように入力します。

```
!no echo
show service S1 | grep -u ":"
echo "The line is: ${UGREP}"
```

次のように出力されます。

```
The line is: Weight: 1       Load: 255
```

デフォルトでは、この出力は検索条件を満たす **show service** の出力画面の最終行ですが、この行だけが条件に一致するわけではありません。たとえば、検索条件を満たす最初の行など、特定の行を検索するには、*[n]* オプションを使用します。たとえば、次のように入力します。

```
!no echo
show service S1 | grep -u1 ":"
echo "The line is: ${UGREP}"
```

次のように出力されます。

```
The line is: Name: S1           Index: 1
```

これは、検索条件を満たす最初の行です。-u1 オプションによって、コロン（:）を検索し、最初に一致した検索結果を変数 UGREP に格納するように指示しています。

grep コマンドによる STATUS

grep コマンドが戻す STATUS コードの値は、一致した検索結果の数に等しくなります。その他のスクリプト コマンドは、いずれも正常終了時には 0 を返すため、違いに注意してください。grep コマンドが返す STATUS は他のコマンドの場合とはまったく異なり、検索結果が 14 件であれば 14、検索結果がなければ 0 になります。

前の項と同じ方法で文字列「Keepalive」を検索すると、検索結果は 1 となります。この場合、変数 STATUS の値は 1 に設定されます。

grep コマンドから返されるステータス コードを **while** ループと組み合わせると、すべての検索結果を 1 行ずつ確認できます。

そのようなスクリプトの一例を次に示します。

```
show service S1 | grep ":"
set endIndex "${STATUS}"
set index "1"
while index "LTEQ" "${endIndex}"
    show service S1 | grep -u${index} ":"
    echo "${UGREP}"
    modify index "++"
endbranch
```

socket コマンド

プロトコルの階層構造を効果的に構築するには、スクリプトのキープアライブで **socket** コマンドを使用します。socket コマンドでは、ASCII コードまたは 16 進数でのデータ送受信が可能です。オプションとして **raw** キーワードを持つ各コマンドでは、データが標準の ASCII から 16 進数に変換されます。たとえば、abcd は ASCII コードでは 61626364、16 進数では 0x61 0x62 0x63 0x64 になります。

socket connect

特定の IP アドレスやポートとの TCP 接続ハンドシェイク（SYN-SYNACK...）、またはホストやポートの予約による UDP 接続を実行するには、**socket connect** コマンドを使用します。ソケットの値は、スクリプト内の変数 ${SOCKET} に格納されます。

このコマンドのシンタックスは次のとおりです。

socket connect host *ip_address* **port** *number* **[tcp {***timeout***} {session}** {**nowait**}|**udp** {**session**}**]**

（注）　1 台の CSS で実行される各種のスクリプトは、最大で 64 のソケットを並行して開くことができます。

このコマンドのオプションと変数は次のとおりです。

- **host**：キーワード。このキーワードの直後には、リモート CSS のホスト名または IP アドレスを必ず指定します。
- *ip_address*：リモート CSS のホスト名または IP アドレス
- **port**：キーワード。このキーワードの直後には、接続をネゴシエートするポートを必ず指定します。
- *number*：接続をネゴシエートするポート番号
- **tcp**：TCP 接続を指定するキーワード
- **udp**：UDP 接続を指定するキーワード

- *timeout*：ネットワーク接続を確立するまでのタイムアウト（ミリ秒）。接続が正しく確立される前にタイムアウトに達すると、接続は失敗します。UDP はコネクションレス型プロトコルであるため、このタイムアウトは TCP 接続だけに適用されます。1 ～ 60000 ミリ秒（1 ミリ秒～ 60 秒）を指定します。デフォルトは 5000 ミリ秒（5 秒）です。
- **session**：セッション終了までソケットを開いたまま保持するキーワード。スクリプトがセッション中に開いたソケットを閉じない場合、そのソケットはログアウトするまで開いたまま保持されます。
- **nowait**：最初にデータが集約されるのを待たずに、ソケットですぐにデータを送信するキーワード

socket send

確立済みの TCP 接続を介してデータを送信するには、**socket send** コマンドを使用します。**socket send** コマンドを実行すると、10KB 受信バッファ内に現在格納されているデータがすべてクリアされます。

このコマンドのシンタックスは次のとおりです。

socket send *socket_number "string"* {**raw** | **base64**}

- *socket_number*：ソケットファイル記述子（整数形式）。この記述子は **socket connect** コマンドから返されます。
- *string*：引用符で囲んだ 128 文字以内のテキスト文字列
- **raw**：文字列値を単純な文字列ではなく 16 進バイトとして解釈するようにソフトウェアに指示するオプションのキーワード。たとえば、0D0A は 0x0D 0x0A（CR+LF）に変換されます。
- **base64**：文字列を接続を介して送信する前に、Base-64 方式でエンコードする。このオプションは、パスワード保護された Web サイトに接続する際の HTTP 基本認証に役立ちます。

socket receive

ソケットの 10KB の内部バッファをリモート ホストからのデータで満たすには、**socket receive** コマンドを使用します。バッファがいっぱいになると、このコマンドによってバッファがロックされ、新しいデータはバッファに格納されなくなります。この 10KB のバッファ内に存在するすべてのデータは、**socket inspect** コマンドを使用して標準出力に出力できます。

（注）　10KB の内部バッファの古いデータは、新しいデータを保存する前にソフトウェアによってすべて削除されます。

このコマンドのシンタックスは次のとおりです。

socket receive *socket_number* {*timeout*} {**raw**}

このコマンドのオプションと変数は次のとおりです。

- *socket_number* : ソケット ファイル記述子（整数形式）。この記述子は **socket connect** コマンドから返されます。
- *timeout* : スクリプトが内部の 10KB バッファをロックして実行を再開するまで CSS ソフトウェアに待たせる時間をミリ秒で指定するオプションのタイムアウト値。1 ～ 15,000 の整数を入力します。デフォルトは 100 ミリ秒です。
- **raw**: 文字列値を単純な文字列ではなく 16 進バイトとして解釈するようにソフトウェアに指示するオプションのキーワード。たとえば、0D0A は 0x0D 0x0A（CR+LF）に変換されます。

socket waitfor

ソケットの 10KB の内部バッファをリモート ホストからのデータで満たすには、**socket waitfor** コマンドを使用します。このコマンドは、引数 *string* で指定された値が見つかるとただちに制御を返す点を除けば、**socket receive** コマンドと同じです。CSS は指定の文字列を見つけると、${STATUS} 値として 0（成功）を返します。それ以外の場合は 1 を返します。取り込まれたデータは **socket inspect** コマンド（次の項参照）で表示できます。

このコマンドのシンタックスは次のとおりです。

socket waitfor *socket_number* [**anything** {*timeout*}|*“string”* {*timeout*} {**case-sensitive**} {**offset** *bytes*} {**raw**}]

このコマンドのオプションと変数は次のとおりです。

- *socket_number*：ソケット ファイル記述子（整数形式）。この記述子は **socket connect** コマンドから返されます。
- **anything**：タイムアウトまでの期間内に着信データを検出すると、ただちに制御を戻すように指定する。この場合、コマンドはデータを検出するとただちに制御を返し、タイムアウト終了まで待ちません。
- *timeout*：引数 *string* で指定された値が見つかるまで CSS に待たせる時間（ミリ秒）を指定するオプションのタイムアウト値。1 ～ 15000 ミリ秒の整数を入力します。デフォルトは 100 ミリ秒です。
- *string*：${STATUS} の値を 0 にするために、CSS が検出すべき文字列。CSS が文字列を検出すると、このコマンドはただちに制御を返し、引数 *integer* で指定されたタイムアウト終了まで待ちません。
- **case-sensitive**：文字列比較で大文字と小文字を区別するように指定するオプションのキーワード。たとえば、User と user は互いに区別されます。
- **offset** *bytes*：受信データの先頭位置から、文字列の検索を開始する位置までのオフセット（バイト単位）を指定するオプションのキーワードと値。たとえば、検索すべき文字列値として a0、オフセットを 10 にそれぞれ指定した場合には、受信データの開始位置から 10 バイト後にある a0 が CSS によって検索されます。
- **raw**：文字列値を単純な文字列ではなく 16 進バイトとして解釈するようにソフトウェアに指示するオプションのキーワード。たとえば、0D0A は 0x0D 0x0A（CR+LF）に変換されます。

socket inspect

ソケットの内部データ バッファに実際のデータがあるかどうかを検査するには、**socket inspect** コマンドを使用します。データが見つかった場合、受信したデータの最後の 10KB が標準出力に表示されます。出力される文字が表示不能な場合は、便宜上ピリオド（.）で置き換えられます （次の *pretty* 引数の下の例を参照してください）。

このコマンドのシンタックスは次のとおりです。

socket inspect *socket_number* {**pretty**}{**raw**}

このコマンドのオプションと変数は次のとおりです。

- *socket_number*：ソケット ファイル記述子（整数形式）。この記述子は **socket connect** コマンドから返されます。
- **raw**：単純な文字列ではなく 16 進バイトで文字列値を表示する。たとえば、ABCD の代わりに 41424344（相当する 1 バイト長の 16 進数）が標準出力に出力されます。
- **pretty**：データの各バイトを 16 進数と ASCII の両方で表して各行を出力する。各行に最大 16 バイトを出力できます。たとえば、0x41 0x42 0x43 0x44 0x10 0x05　ABCD.. と表示されます。

（注）　keepalive スクリプトで **socket inspect** コマンドを使用する場合は、サービスのキープアライブ タイプを設定するときにコマンド行で “use-output” オプションを指定する必要があります。

socket disconnect

CSS がデータの送信を完了したことがわかるように、RST（リセット）をリモート ホストに送信してリモート ホストとの接続を閉じるには、**socket disconnect** コマンドを使用します。

このコマンドのシンタックスは次のとおりです。

socket disconnect *socket_number* {**graceful**}

- *socket_number*：ソケット ファイル記述子（整数形式）。この記述子は **socket connect** コマンドから返されます。
- **graceful**：RST ではなく FIN（完了）をリモート ホストに送信して接続を適切に閉じる。

ソケット管理

CSS では、最大 64 のソケットを並行して開く（使用する）ことができます。**socket connect** コマンドを発行した後、そのソケットのファイル記述子（変数 ${SOCKET} に保存）を対象として **socket disconnect** コマンドを発行しない場合、そのソケットのファイル記述子を引数 *socket_number* に指定して **socket disconnect** コマンドを発行するまで、そのソケットは開いたままになります。

スクリプト内で開いたソケットは、**socket connect** コマンドに引数 *session* を渡している場合を除き、スクリプトの終了時に自動的に閉じます。セッション内で開いたソケットは、そのセッションの終了時（ユーザのログアウト時）に閉じられます。

現在使用中のすべてのソケット ファイル記述子のリストを表示する場合に、**show sockets** コマンドを使用します。

表 8-2 に、**show sockets** コマンドで表示されるフィールドについて説明します。

表 8-2　show sockets コマンドのフィールド

フィールド	説明
Socket ID	ソケット ファイル記述子
Remote Host:Port	接続先のホスト アドレスとポートのペア
Protocol	接続に指定されているプロトコル（TCP または UDP）
User	**show lines** コマンドで表示される回線 ID
Time	記述子のソケット ファイルが開かれている時間

（注）　リモート ホストがタイムアウトするか、またはソケットを閉じると、ソケットのアーキテクチャによりソケットがクリーンアップされ、使用中のソケットのリストから削除されます。このクリーンアップは、リモート ホストがすでに閉じているソケットで別の転送を行った後にだけ実行されます。それ以外の場合、ソケットはアイドル状態のままです。

データ バッファの 10KB のデータを一度に取得するには、**socket receive** コマンドを実行します。このバッファは、別の **socket receive** コマンドまたは **socket waitfor** コマンドを発行するまで同じ状態のままです。バッファは、別のコマンドを発行した時点で消去され、リモート ホストからの別のデータで再度満たされます。各ソケット記述子（**socket connect** コマンドで作成）には、専用の 10KB のバッファがあります。

（注）　**socket send** コマンドを実行すると、10KB バッファ内に現在格納されているデータもすべてクリアされます。詳細については、「socket send」を参照してください。

スクリプトの表示

CSS の script ディレクトリに格納されているスクリプトのリストまたは特定のスクリプトの内容（行番号を含めることもできます）を表示するには、**show script** コマンドを使用します。このコマンドは、スーパーユーザ モードとすべての設定モードで使用可能です。このコマンドのシンタックスは次のとおりです。

show script {*filename* {**line-numbers**}}

このコマンドの変数とオプションは次のとおりです。

- *filename*：表示する有効なスクリプト ファイルの名前。大文字小文字を区別して、32 文字以内のテキスト文字列を引用符で囲まずに入力します。
- **line-numbers**：スクリプト内の各行の行番号を表示する。

たとえば、CSS の script ディレクトリ内に格納されているすべてのスクリプトのリストを表示するには、次のように入力します。

```
# show script
```

ap-ka-dns キープアライブ スクリプトの内容を、行番号を含めて表示するには、次のように入力します。

```
# show script ap-kal-dns line-numbers
```

スクリプトのアップグレードにあたっての留意事項

CSS ソフトウェアを新しいバージョンにアップグレードする際には、従来のバージョンの /script ディレクトリ内にある変更済みのスクリプト ファイルを、新しいバージョンの /script ディレクトリにすべてコピーする必要があります。この作業を行わないと、スクリプトは以前の /script ディレクトリに残ったままになり、CSS で認識されません。

CSS ソフトウェアをアップグレードする*前*に、次の操作を行ってください。

1. スーパーユーザ モードで **archive script** コマンドを使用して、各スクリプト ファイルをアーカイブします。スクリプトのアーカイブの詳細については、第 1 章「CSS ソフトウェアの管理」を参照してください。
2. CSS ソフトウェアをアップグレードします。CSS ソフトウェア バージョンのアップグレードの詳細については、付録 A「CSS ソフトウェアのアップグレード」を参照してください。
3. スーパーユーザ モードで **restore script** コマンドを使用して、新しいソフトウェア バージョンの /script サブディレクトリにスクリプトを復元します。スクリプトの復元の詳細については、第 1 章「CSS ソフトウェアの管理」を参照してください。

showtech スクリプト

サービス担当が CSS を分析するときに必要な情報を収集するには、**showtech** スクリプトを使用します。このスクリプトの出力には、サービス担当が問題解決のために使用する CSS のステータスと設定情報のセットが表示されます。CSS に接続しているアプリケーションの出力キャプチャ機能を使用して、分析に使用するスクリプトの出力を保存します。

また、次に表示するプロンプトで **y** を入力すると、ファイル log/showtech.out にスクリプトの出力を保存できます。必要な場合は、保存した出力ファイルをコピーして、サービス担当に送付できます。

このスクリプトを実行するには、次のコマンドを入力します。

```
# script play showtech

Save output to disk [y/n]? y
Output will be saved to log/showtech.out
Please wait...
```

n を入力すると、出力は画面に表示されます。

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: showtech
!
! Description:
!       show tech-support script
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

set CONTINUE_ON_ERROR "1"

no terminal more
llama

show clock
show disk
show running-config
flow statistics
show service-internal
show service summary
show service
show system-resources
show dump-status
show core
show circuit all
show arp
show ip route
show phy
show summary
show rule
show group
show ether-errors
show keepalive
show ip stat
show rmon
show bridge status
show bridge forwarding
show interface
show virtual-routers
show critical-services
show redundancy
show chassis inventory
show chassis verbose
show log sys.log tail 200
exit
terminal more

set CONTINUE_ON_ERROR "0"

exit script 0
```

スクリプト キープアライブの例

CSS には、スクリプト化されていないキープアライブでは処理できないキープアライブ操作に必要な処理をサポートするためのスクリプト化されたキープアライブ操作が用意されています。スクリプト化されたキープアライブでの I/O 操作は、サーバへのネットワーク接続のプローブに使用するソケット操作を行う場合と、サーバでのアプリケーションの状態を判断する場合にだけ実行するようにしてください。スクリプト言語は CSS のハード ドライブやフラッシュ ディスクでのファイルの I/O をサポートしますが、スクリプト化されたキープアライブ内ではファイルの I/O 操作を使用しないようにしてください。スクリプト化されたキープアライブ内でファイル I/O 操作を実行すると、サービスに影響を及ぼす場合があります。CLI またはコマンド スケジューラから実行されたスクリプト内では、ファイル システムへのアクセスが許可されます。

以降では、スクリプト キープアライブの例を示します。これらの例は、そのまま使うことも、また用途に応じて修正を加えることもできます。

ソケットを通常の手順でクローズ（FIN）するカスタム TCP スクリプト キープアライブの例

次のスクリプト キープアライブでは、ユーザ指定の TCP ポート上でソケットをオープンし、通常の手順でクローズ（RST ではなく FIN を使用）します。

（注）　**keepalive tcp-close fin** サービス モード コマンドまたは **tcp-close fin** グローバル キープアライブ モード コマンドを使用した場合も、通常の手順でクローズ（RST ではなく FIN を使用）します。

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-tcp-ports
! Parameters: Service Address, TCP Port(s)
!
!Description:

! This script will open and close a socket on the user specified
! ports.
! The close will be a FIN rather than a RST. If one of the ports fails
! the service will be declared down
!
! Failure Upon:
! Not establishing a connection with the host on one of the specified
! ports.
!
! Notes:  Does not use output
!     Will handle out of sockets scenario.
!
! Tested: KGS 12/18/01
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

set OUT-OF-SOCKETS "785"
set NO-CONNECT "774"

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "LT" "2"
  echo "Usage: ap-kal-tcp-ports \'ipAddress tcpPort1 [tcpPort2
tcpPort3...]\'"
  exit script 1
endbranch

set SERVICE "${ARGS}[1]"
!echo "SERVICE = ${ARGS}[1]"
var-shift ARGS

while ${ARGS}[#] "GT" "0"
  set TCP-PORT "${ARGS}[1]"
  var-shift ARGS
  function SOCKET_CONNECT call
! If we're out of sockets, exit and look for sockets on the next KAL
interval
  if RETURN "==" "${OUT-OF-SOCKETS}"
    set EXIT_MSG "Exceeded number of available sockets, skipping until
next interval."
    exit script 0
  endbranch

! Valid connection, look to see if it was good
  if RETURN "==" "${NO-CONNECT}"
```

```
    set EXIT_MSG "Connect: Failed to connect to
${SERVICE}:${TCP-PORT}"
    exit script 1
  endbranch
endbranch

no set EXIT_MSG
exit script 0

function SOCKET_CONNECT begin
  set CONTINUE_ON_ERROR "1"
  socket connect host ${SERVICE} port ${TCP-PORT} tcp 2000
  set SOCKET-STAT "${STATUS}"
  set CONTINUE_ON_ERROR "0"
  socket disconnect ${SOCKET} graceful
  function SOCKET_CONNECT return "${SOCKET-STAT}"
function SOCKET_CONNECT end
```

デフォルトのスクリプト キープアライブ

次に示すスクリプト キープアライブは、CSS の /script ディレクトリに置かれており、リストに続けてそれぞれの定義を示します。

- SMTP キープアライブ
- NetBIOS 名のクエリー（Microsoft ネットワーキング）
- HTTP リスト キープアライブ
- POP3 キープアライブ
- IMAP4 キープアライブ
- Pinglist キープアライブ
- Finger キープアライブ
- Time キープアライブ
- Setcookie キープアライブ
- HTTP 認証キープアライブ
- DNS キープアライブ
- Echo キープアライブ
- HTTP ホスト タグ キープアライブ
- Mailhost キープアライブ
- LDAP キープアライブ

SMTP キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-smtp
! Parameters: HostName
!
! Description:
!  This script will log into an SMTP server and send a 'hello'
!  to make sure the SMTP server is stable and active.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Failure to get a good status code after saying 'hello'
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "NEQ" "1"
    echo "Usage: ap-kal-smtp \'Hostname\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 25 tcp

set EXIT_MSG "Waitfor: Failed"
! Receive the incoming status code 220 "welcome message"
socket waitfor ${SOCKET} "220" 200

set EXIT_MSG "Send: Failed"
! Send the helo to the server
socket send ${SOCKET} "helo ${HostName}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for status code "250" to be returned
socket waitfor ${SOCKET} "250" 200

! We've successfully logged in, the server is up and running.
! The job was done successfully.
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0
```

NetBIOS 名のクエリー（Microsoft ネットワーキング）

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-netbios
! Parameters: Hostname
!
! Description:
!   We will make a netbios name query that we know will be
!   a "negative" response.  RFC-1002 NETBIOS states that a hex
!   value of:
!    0x81 Session Request
!    0x82 Positive Session Response
!    0x83 Negative Session Response
!   We will key off of 0x83 which states we failed, but which
!   also means that the service was stable enough to know that
!   we are not a valid machine on the network.
!   This script will send an encoded message for Session Request
!   (0x81) and will invent a CALLER and a CALLED machine name
!   (Caller = this script and CALLED = Server)
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not receiving a status code 0x83 (negative response)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "1"
    echo "Usage: ap-kal-pop3 \'Hostname\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"

! Connect to the remote host (default timeout)
set EXIT_MSG "Connection failure"
socket connect host ${HostName} port 139 tcp

! Send a Netbios Session Request (0x81) and its required encoded
!values.
! This value will be sent in RAW Hex
set EXIT_MSG "Send: Failure"
socket send ${SOCKET}
810000442045454550454d454d464a434143414341434143414341434143414341434143414341
434100" raw
! Wait for a response code of 0x83
set EXIT_MSG "Waitfor: Failure"
socket waitfor ${SOCKET} "83" raw
no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0
```

HTTP リスト キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-httplist
! Parameters: Site1 WebPage1 Site2 WebPage2 [...]
!
! Description:
!   This script will connect a list of sites/webpage pairs.  The
!   user must simply supply the site, and then the webpage and
!   we'll attempt to do an HTTP HEAD on that page.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not receiving a status code 200 on the HEAD request on any
!      one site.  If one fails, the script fails.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "LT" "2"
    echo "Usage: ap-kal-httplist \'WebSite1 WebPage1 WebSite2 WebPage2
...'"
    exit script 1
endbranch

while ${ARGS}[#] "GT" "0"
    set Site "${ARGS}[1]"
    var-shift ARGS

    if ${ARGS}[#] "==" "0"
     set EXIT_MSG "Parameter mismatch: hostname present but webpage
was not"

     exit script 1
    endbranch
    set Page "${ARGS}[1]"
    var-shift ARGS
    no set EXIT_MSG
    function HeadUrl call "${Site} ${Page}"
endbranch
exit script 0
function HeadUrl begin

echo "Getting ${ARGS}[1] from Site ${ARGS}[2]\n"
! Connect to the remote Host
set EXIT_MSG "Connect: Failed to connect to ${ARGS}[1]"
socket connect host ${ARGS}[1] port 80 tcp

! Send the head request
```

```
set EXIT_MSG "Send: Failed to send to ${ARGS}[1]"
socket send ${SOCKET} "HEAD ${ARGS}[2] HTTP/1.0\n\n"

! Wait for the status code 200 to be given to us
set EXIT_MSG "Waitfor: Failed to wait for '200' on ${ARGS}[1]"
socket waitfor ${SOCKET} " 200 "

no set EXIT_MSG
socket disconnect ${SOCKET}

function HeadUrl end
```

POP3 キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-pop3
! Parameters: HostName UserName Password
!
! Description:
!   This script will connect to a POP3 server and login with the
!   username/password pair specified as argument 2 and 3.  After which
!   it will log out and return.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to log in with supplied username/password.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-pop3 \'Hostname UserName Password\'"
    exit script 1
endbranch
! Defines:
set HostName "${ARGS}[1]"
set UserName "${ARGS}[2]"
set Password "${ARGS}[3]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 110 tcp
set EXIT_MSG "Waitfor: Failed"
! Wait for the OK welcome message for 200ms
socket waitfor ${SOCKET} "+OK" 200

set EXIT_MSG "Send: Failed"
! Send the username to the host
```

```
socket send ${SOCKET} "USER ${UserName}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "+OK" 200

set EXIT_MSG "Send: Failed"
! Send the password
socket send ${SOCKET} "PASS ${Password}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "+OK" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0
```

IMAP4 キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-imap4
! Parameters: HostName UserName Password
!
! Description:
!   This script will connect to a IMAP4 server and login with the
!   username/password pair specified as argument 2 and 3.  After which
!   it will log out and return.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to log in with supplied username/password.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-imap4 \'Hostname UserName Password\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"
set UserName "${ARGS}[2]"
```

```
set Password "${ARGS}[3]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 143 tcp

set EXIT_MSG "Waitfor: Failed"
! Wait for the OK welcome message for 600ms
socket waitfor ${SOCKET} "OK" 600

set EXIT_MSG "Send: Failed"
! Send the username to the host
socket send ${SOCKET} "a1 LOGIN ${UserName} ${Password}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "a1 OK" 200

set EXIT_MSG "Send: Failed"
! Send the password
socket send ${SOCKET} "a2 LOGOUT\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "a2 OK" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0
```

Pinglist キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-pinglist
! Parameters: HostName1 HostName2 HostName3, etc.
!
! Description:
!   This script is designed to ping a list of hosts that the user
!   passes in on the command line.
!
! Failure Upon:
!   1. Not being able to ping any one of the hosts in the list
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "LT" "1"
    echo "Usage: ap-kal-pinglist \'HostName1 HostName2 HostName3
...\'"
    exit script 1
endbranch

while ${ARGS}[#] "GT" "0"
    set Host "${ARGS}[1]"
    var-shift ARGS
    function PingHost call "${Host}"
endbranch

no set EXIT_MSG
exit script 0

function PingHost begin

! Ping the first host
ping ${ARGS}[1] | grep -u Success
if STATUS "NEQ" "0"
    show variable UGREP | grep 100
    if STATUS "==" "0"
     set EXIT_MSG "Ping: Failure to ping ${ARGS}[1]"
     exit script 1
    endbranch
endbranch

function PingHost end
```

Finger キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-finger
! Parameters: HostName UserName
!
! Description:
!   This script will connect to the finger server on the remote
!   host.  It will query for the UserName and receive the
!   information back.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to send/receive data to the host
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "2"
    echo "Usage: ap-kal-finger \'Hostname UserName\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"
set UserName "${ARGS}[2]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 79 tcp

set EXIT_MSG "Send: Failed"
! Send the username to "finger"
socket send ${SOCKET} "${UserName}\n"
set EXIT_MSG "Waitfor: Failed"
! Wait for data for 100ms (default)
socket waitfor ${SOCKET} "${UserName}"

no set EXIT_MSG
! If the data came in, then we are good to quit
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0
```

Time キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-time
! Parameters: HostName
!
! Description:
!   This script will connect to a remote host 'time' service on
!   port 37 and get the current time.  This script currently works
!   strictly with TCP.  [Ref. RFC-868]
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to receive incoming data
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "NEQ" "1"
                echo "Usage: ap-kal-time \'Hostname\'"
                exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 37 tcp 2000

set EXIT_MSG "Receive: Failed"
! waitfor any data for 2000ms
socket waitfor ${SOCKET} anything 2000


! If the data came in, then we are good to quit
socket disconnect ${SOCKET}

no set EXIT_MSG

exit script 0
```

Setcookie キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-setcookie
! Parameters: HostName WebPage cookieString
!
! Description:
!   This script will keepalive a WWW server that is setting
!   cookies in the HTTP response header.  The header value
!   looks like this:
!   Set-Cookie: NAME=VALUE
!
!   The user will be responsible for sending us the name & value
!   in a string like "mycookie=myvalue" so that we can compare
!   the incoming Set-Cookie: request.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to receive the cookie
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-setcookie \'Hostname WebPage cookieString\'"
    echo "(Where cookieString is a name=value pair like
\'mycookie=myvalue\')"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"
set WebPage "${ARGS}[2]"
set CookieData "${ARGS}[3]"

! Connect to the remote host (use default timeout)
set EXIT_MSG "Connection Failed"
socket connect host ${HostName} port 80 tcp

! send our request to the host
set EXIT_MSG "Send: Failure"
socket send ${SOCKET} "GET ${WebPage} HTTP/1.0\n\n"

! Wait for the cookie to come in
set EXIT_MSG "Waitfor: Failure"
socket waitfor ${SOCKET} "${CookieData}"

! Done
no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0
```

HTTP 認証キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-httpauth
! Parameters: HostName WebPage Username-Password
!
! Description:
!   This will keepalive an authentication connection by building
!   a get request with the Authentication field filled with the
!   Username-Password string formatted like so: "bob:mypassword"
!   This is critical to make the authentication base64 hash work
!   correctly.
!
! Note: This script authentication is based on HTTP AUTHENTICATION
!       RFC-2617.  Currently only supported option is "Basic"
!       authentication using base64 encoding.  "Digest" Access is
!       not supported at this time.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able authenticated with the Username-Password
!      (not being given a status code of "200 OK"
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-httpauth \'Hostname WebPage
Username:Password\'"
    echo "(Ie. ap-kal-httpauth \'192.168.1.1 /index.html
bob:mypassword\')"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"
set WebPage "${ARGS}[2]"
set UserPass "${ARGS}[3]"

! Connect to the remote Host
set EXIT_MSG "Connection Failure"
socket connect host ${HostName} port 80 tcp

! Send the GET request for the web page, along with the authorization
! This builds a header block like so:
!
! GET /index.html HTTP/1.0\r\n
! Authorization: Basic bGFiOmxhYnRlc3Qx\r\n\r\n

set EXIT_MSG "Send: Failed"
socket send ${SOCKET} "GET ${WebPage} HTTP/1.0\n"
socket send ${SOCKET} "Authorization: Basic "
```

```
socket send ${SOCKET} "${UserPass}" base64
socket send ${SOCKET} "\n\n"

! Wait for a good status code
set EXIT_MSG "Waitfor: Failed"
socket waitfor ${SOCKET} "200 OK"

no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0
```

DNS キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-dns
! Parameters: Server DomainName
!
! Description:
!   This script will resolve a domain name from a specific DNS
!   server.  This builds a UDP packet based on RFC-1035
!
! Failure Upon:
!   1. Not resolving the hosts's IP from the domain name
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "1"
    echo "Usage: ap-kal-finger \'Hostname\'"
    exit script 1
endbranch

set HostName "${ARGS}[1]

! Connect to the remote host
set EXIT_MSG "Connection failed"
socket connect host ${HostName} port 53 udp

! This may require a little explanation.  Since we just want to see
! if the DNS server is alive we will send a simple DNS Query.  This
! query is hard coded in hexadecimal and sent raw to the DNS server.
! The DNS request has a 12 byte header (as seen for the first 12 bytes
! of hex) and then a DNS name (ie. www.cisco.com).  Lastly it follows
! with some null termination and a few bytes representing query type.
! See RFC-1035 for more.
set EXIT_MSG "Send: failure"
socket send ${SOCKET}
"000201000001000000000000037777770563697363 6f03636f6d0000010001" raw

! Receive some unexplained response.  We don't care what it is because
! an unstable DNS server or a non-existent one would probably not send
```

```
! us any data back at all.

set EXIT_MSG "Receive: Failed to receive data"
socket receive ${SOCKET}

no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0
```

Echo キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-echo
! Parameters: HostName [ udp | tcp ]
!
! Description:
!   This script will send a TCP or UDP echo (depending on what the
!   user has passed to us) that will echo "Hello Cisco" to the
!   remote host, and expect it to come back.  The default protocol
!   is TCP.
!
! Failure Upon:
!   1. Not establishing a connection with the host (TCP Only).
!   2. Not being able to retrieve an echoed message back
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "NEQ" "2"
    if ${ARGS}[#] "NEQ" "1"
     echo "Usage: ap-kal-echo \'Hostname [ udp | tcp ]\'"
     exit script 1
    endbranch
endbranch
! Defines:
set HostName "${ARGS}[1]"
set nProtocol "tcp"

! See if the user has specified a protocol
if ${ARGS}[#] "==" "2"
    ! The user specified a protocol, so reset the value
    set nProtocol "${ARGS}[2]"
endbranch

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 7 ${nProtocol}

set EXIT_MSG "Send: Failed"
! Send the text to echo...
```

```
socket send ${SOCKET} "Hello Cisco!\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for the reply from the echo (should be the same)
socket waitfor ${SOCKET} "Hello Cisco!" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.

socket disconnect ${SOCKET}
no set EXIT_MSG
exit script 0
```

HTTP ホスト タグ キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-httptag
! Parameters: HostName WebPage HostTag
!
! Description:
!   This script will connect to the remote host and do an HTTP
!   GET method upon the web page that the user has asked for.
!   This script also adds a host tag to the GET request.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not receiving an HTTP status "200 OK"
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-httptag \'Hostname WebPage HostTag\'"
    exit script 1
endbranch
! Defines:
set HostName "${ARGS}[1]"
set WebPage "${ARGS}[2]"
set HostTag "${ARGS}[3]"

! Connect to the remote Host
set EXIT_MSG "Connection Failure"
socket connect host ${HostName} port 80 tcp

! Send the GET request for the web page
set EXIT_MSG "Send: Failed"
socket send ${SOCKET} "GET ${WebPage} HTTP/1.0\nHost: ${HostTag}\n\n"

! Wait for a good status code
set EXIT_MSG "Waitfor: Failed"
```

```
socket waitfor ${SOCKET} "200 OK"

no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0
```

Mailhost キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-mailhost
! Parameters: HostName UserName Password
!
! Description:
!   This script will check the status on a mailhost.  The mailhost
!   should be running a POP3 and SMTP service.  We will attempt
!   to keepalive both services, and if one goes down we will report
!   an error.
!
! Failure Upon:
!   1. Not establishing a connection with the host running an SMTP
     service.
!   2. Not establishing a connection with the host running a POP3
     service.
!   3. Failure to get a good status code after saying 'hello' to SMTP.
!   4. Failure to login using POP3.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-pop3 \'Hostname UserName Password\'"
    echo "(For checking an SMTP and POP3 service)"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"
set UserName "${ARGS}[2]"
set Password "${ARGS}[3]"

!!!!! SMTP !!!!!!

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 25 tcp

set EXIT_MSG "Waitfor: Failed"
! Receive the incoming status code 220 "welcome message"
socket waitfor ${SOCKET} "220" 200

set EXIT_MSG "Send: Failed"
! Send the hello to the server
```

```
socket send ${SOCKET} "helo ${HostName}\n"
set EXIT_MSG "Waitfor: Failed"
! Wait for status code "250" to be returned
socket waitfor ${SOCKET} "250" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.
socket disconnect ${SOCKET}

!!!!! POP3 !!!!!!

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 110 tcp

set EXIT_MSG "Waitfor: Failed"
! Wait for the OK welcome message for 200ms
socket waitfor ${SOCKET} "+OK" 200

set EXIT_MSG "Send: Failed"
! Send the username to the host
socket send ${SOCKET} "USER ${UserName}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "+OK" 200
set EXIT_MSG "Send: Failed"
! Send the password
socket send ${SOCKET} "PASS ${Password}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "+OK" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0
```

LDAP キープアライブ

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-ldap
! Parameters: HostName
!
! Description:    "Lightweight Directory Access Protocol v3"
!   This script will connect to an LDAP server and attempt to
!   "bind request" to the server.  Once the server gives a
!   positive response we will disconnect (RFC-2251).
!
! Bind Response Code we will search for is: 0x0a 0x01 0x00
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Failure to receive the above response code.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "NEQ" "1"
    echo "Usage: ap-kal-ldap \'Hostname\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 389 tcp 2000

set EXIT_MSG "Send: Failure"
! Send a Bind Request to the remote host.  This is simply a standard !
"capture" of a bind request in hex.  This should work for all standard
! version 3 LDAP servers. socket send ${SOCKET}
"300c020102600702010204008000" raw

set EXIT_MSG "Receive: Failure"
! Expect to receive a standard response from the host.  This should !
be equal to a SUCCESS response code: socket waitfor ${SOCKET} "0a0100"
2000 raw


set EXIT_MSG "Send: Failure"
! Send an exit "Unbind Request" to the remote host so that they ! are
not left hanging. socket send ${SOCKET} "30050201034200" raw

no set EXIT_MSG
socket disconnect ${SOCKET}

exit script 0
```