**1) 最前面表示キー**

それではまず、**最前面表示キー**について説明します。

このキーの役割はとてもシンプルです。  
「電卓をディスプレイの最前面に固定する。そして、もう一度押すと解除する」  
これが基本の仕様です。

コードでは、**行474から480**にある btnTopMost\_Click メソッドで実装されています。  
このメソッドの中で特に重要なのが、

this.TopMost = !this.TopMost;

という一行です。

ここで TopMost プロパティを切り替えているわけです。  
true ならウィンドウは常に最前面、false なら通常の状態に戻る。  
つまりボタンを押すたびに、**固定と解除が交互に切り替わる**仕組みになっています。

さらに、単に機能が切り替わるだけではありません。  
ユーザーが **今どういう状態なのかを視覚的に分かるようにする工夫** がされています。

それが、メソッドの最後で呼ばれている UpdateTopMostButtonColor() です。  
この処理は **行1240から1252** に書かれていて、ボタンの背景色を切り替えています。  
具体的には、固定状態のときは **水色（LightBlue）**、解除されているときは **標準のボタン色（SystemColors.Control）**。

この仕組みによって、ユーザーは **「ボタンの色が変わった」=「最前面に固定されている」** とすぐに理解できます。

例えば、資料を見ながら電卓を使うときに、うっかりウィンドウが裏に隠れると不便ですよね。  
この機能を使えば常に画面上に表示され、しかもボタンの色で状態を確認できるので、安心して計算に集中できるわけです。

まとめると、

* TopMost プロパティを切り替えるだけで実現できるシンプルな仕組み。
* さらにボタンの色を変えて、状態を一目で分かるようにしている。  
  この2つの組み合わせで、実用性とユーザビリティを両立している、というのがポイントです。

**2) 途中計算表示欄**

続いてご説明するのは、**途中計算表示欄**です。

この欄は、現在入力中の数値や演算子を表示する役割を持っています。  
例えば「123 × 5 =」と入力した場合には、「123 ×」という式の進行状況がここに表示されます。

では、どのように制御されているのでしょうか。

キーとなるのは、**行1301から1320**の IsInputValid メソッドです。  
ここでは、入力できる桁数を制御しています。

使っている定数は2つ。

* Constants.Numeric.MAX\_DISPLAY\_DIGITS = 16
* Constants.Numeric.MAX\_FRACTIONAL\_DIGITS = 17

このルールはとても重要です。  
整数を含む通常の数値は最大で16桁まで入力できます。  
一方、小数点以下が続く「0.」で始まる数値の場合は、最大で17桁まで入力が可能です。

なぜ17桁なのか。  
理由は「先頭の0を含めても、実質的には有効桁数16桁に収めたい」からです。  
「0.12345678901234567」のようなケースがそうですね。

これにより、途中計算表示欄に並ぶ数値は、2つ揃っても34桁を超えないようになっています。  
つまり、画面の中に必ず収まる範囲で表示が完結するわけです。

さらに、もう一つ重要な仕組みがあります。  
**イコール直後に新しい数値を入力した場合、途中計算表示欄をクリアする**という処理です。

これは **行1983から1990** の HandleInitialState メソッドで実装されています。  
具体的には、ExpressionEndsWithEqual() で末尾が「=」かどうかを確認し、もしそうなら ResetCalculatorState() を呼び出して表示をリセットします。

この処理があるおかげで、例えば「123 × 5 = 615」と計算した後に「3」と入力すると、新しい式「3」からスタートできるのです。  
もしこのリセットがなかったら、「6153」のように数字がくっついてしまい、とても使いにくいですよね。

つまり途中計算表示欄は、

* 入力桁数を16桁または17桁に制御する。
* イコール直後はリセットして新しい式を始められる。  
  この2点を満たすことで、ユーザーが常に分かりやすく操作できるようになっています。

**3) 計算結果表示欄**

次に、**計算結果表示欄**についてです。

この欄は、ユーザーが最も注目する部分ですね。  
答えがどう出るのか、桁数やフォーマットはどうなるのか。  
仕様としては次のようになっています。  
「整数部と小数部を合わせて最大16桁。ただし小数点以下が続く 0.123... 形式の場合のみ17桁まで許容する」

この仕様を支えているのが、**行1819から1855**の RoundResult メソッドです。  
この中では、まず整数部の桁数をカウントし、それに応じて小数部の桁数を調整します。  
合計で16桁以内に収める、というのが基本ルールです。

例えば「1234567890123456.78」のような値を計算すると、整数部だけで16桁ありますから、小数部は表示できません。  
逆に「0.12345678901234567」の場合、小数部16桁まで許容されます。整数の「0」と合わせて合計17桁というわけです。

次に文字サイズの調整です。  
大きな数字を表示するとき、枠に入りきらなかったらどうなるでしょうか。  
この場合、自動で文字サイズを縮小する仕組みが働きます。

それを担当しているのが、**行1903から1956**の AutoFitResultFont メソッドです。  
内部では TextRenderer.MeasureText を使って、現在の文字列の描画幅を計算します。  
もし枠からはみ出す場合は、Constants.FontSize.SIZE\_EPSILON の値だけ少しずつ縮小していきます。  
最終的には Constants.FontSize.MIN\_LIMIT = 14f まで小さくすることができます。

これにより、たとえ大きな桁数の数字でも、画面に必ず収まるようになります。  
ユーザーは「数字が切れて見えない」という不便を感じずに済むのです。

さらに表示形式の工夫もあります。  
一つ目は「整数部を3桁ごとにカンマで区切る」こと。  
これは **行1605から1626** の UpdateTextResultWithCommas で実装されています。  
内部では InsertCommasIfNeeded を呼び出し、必要に応じて「1,234,567」のように整形します。

二つ目と三つ目は「極端に大きい数や小さい数は指数表記に切り替える」という仕様です。  
この処理は **行1756から1806** の FormatNumberForDisplay に書かれています。  
具体的には、数値の絶対値が Constants.Numeric.SCI\_LARGE\_THRESHOLD = 1e16 を超えると指数表記になります。  
指数の文字列を実際に生成するのは **行1711から1750** の FormatExponential メソッドです。

例えば「9,999,999,999,999,999」を超えると、通常表示ではなく「1e+16」のような形式に切り替わります。

ここでポイントを整理すると、計算結果表示欄は：

* 有効桁数を16桁、または0.で始まる場合は17桁までに制御する。
* フォントサイズを自動で縮小して必ず枠に収める。
* 整数部にはカンマを入れて読みやすくする。
* 桁が大きくなりすぎたら指数表記に切り替える。

このように、**見やすさと正確さのバランスを取った仕組み**になっています。

**4) パーセントキー（%）**

次にご説明するのは、**パーセントキー**です。

このキーの仕様は、やや複雑です。  
単純に「入力値を100で割る」だけではなく、**前の演算子によって意味が切り替わる**のが大きな特徴です。

まずコードの場所ですが、メイン処理は **行843から960の OnPercentButton メソッド** にあります。  
このメソッドの最初で呼ばれているのが、**行1548から1551の CalculatePercent** です。  
ここでは、入力値に Constants.Numeric.PERCENT\_MULTIPLY を掛けています。  
定数の値は 0.01 ですので、単純に「値を100分の1にする」処理です。

では、実際の動作を見てみましょう。

**ケース1：掛け算や割り算のとき**

掛け算や割り算の直後に % を押すと、右辺の値がそのまま 1/100 に置き換わります。  
例えば「200 × 10 %」と入力すると、右辺の 10 が 0.1 に変換されます。  
つまり式は「200 × 0.1」となり、結果は 20 です。

**ケース2：加算や減算のとき**

ここが特別です。  
加算や減算の直後に % を押すと、右辺の数値を「左辺に対する割合」として扱います。

コード上では **行874から892** の分岐で実装されています。  
ここで replacedB = m\_firstValue \* percent; という処理が行われています。

例えば「200 + 10 %」と入力したとしましょう。  
右辺の「10%」は「200 × 0.1 = 20」と置き換えられます。  
つまり、式全体は「200 + 20」となり、最終結果は 220 になります。

**状態管理の工夫**

さらに OnPercentButton では、% が押されたことを記録するフラグが2つ使われています。

* m\_lastActionWasPercent
* m\_lockRhsAfterAutoOp

この2つのフラグがあるおかげで、例えば % を押した直後に CE を押すと「右辺だけ消す」という挙動が実現できます。  
実際の Windows 標準電卓と同じ動きです。

**まとめ**

パーセントキーは：

* 掛け算・割り算 → 単純に右辺を 1/100 にする。
* 加算・減算 → 左辺に対する割合に変換する。
* 専用フラグで CE や連続入力にも対応。

と、演算子の種類に応じて解釈が変わる仕様になっています。

**5) クリアエントリーキー（CE）**

続いては、\*\*クリアエントリーキー（CE）\*\*です。

仕様としては「計算結果表示欄の数値を削除して 0 に戻す。ただし途中計算表示欄は保持される」という動作になっています。

コードは **行965から1011の OnClearEntryButton メソッド**にまとまっています。

最初に呼ばれるのが DisplayZeroResult() です。  
この処理は 、textResult.Text を「0」に戻すと同時に、内部変数 m\_displayValue を 0 に更新しています。

**挙動の具体例**

では、いくつか例を見てみましょう。

* 例1：「23 + 6」と入力して CE を押す。  
  　→ 途中計算表示欄は「23 +」のまま残り、結果表示欄だけが「0」になります。
* 例2：その後に「5」と入力する。  
  　→ 途中計算欄は「23 +」のままで、結果表示欄に「5」が入ります。
* 例3：逆に CE の後に「−」を押す。  
  　→ 演算子が置き換わり、途中計算欄は「23 −」となり、結果欄は左辺の「23」に戻ります。

**%キーとの組み合わせ**

ここで大事なのが、%キーとの組み合わせです。

コードの中には **%専用の分岐処理**があります。  
直前の操作が % の場合、CE を押すと右辺を削除して左辺だけ残す動作になります。  
これは m\_lastActionWasPercent と m\_lockRhsAfterAutoOp というフラグで制御されています。

実際の動作を確認すると、例えば「200 + 10 %」と入力後に CE を押すと、途中計算欄は「200 +」に戻り、右辺だけが消えます。

**まとめ**

クリアエントリーキー（CE）は、

* 「右辺だけをクリア」するキー。
* 途中式は残すため、継続して新しい右辺を入力できる。
* % と組み合わせると「右辺消去」が正しく機能する。

このように、**部分的なリセット**を実現している点が特徴です。

**6) クリアキー（C）**

次にご説明するのは、\*\*クリアキー（C）\*\*です。

仕様は CE とよく似ていますが、違いは「全体を初期化する」という点です。  
Cキーを押すと、途中計算表示欄も、計算結果表示欄も、両方とも消去されます。  
そして、結果表示欄には「0」が戻ります。

コードは **行1074から1076の OnClearButton メソッド**に実装されています。  
ここでは ResetAllState() が呼ばれています。

この ResetAllState() では以下の初期化を行っています。

* 数値の初期化：m\_firstValue、m\_secondValue、m\_displayValue をすべて 0 に戻す。
* テキストの初期化：textResult.Text を「0」に、textExpression.Text を空にする。
* フラグの初期化：m\_textOverwrite、m\_numDot、m\_isErrorState などをリセット。
* フォントの初期化：ResetFonts() を呼び出し、文字サイズを基準値に戻す。

**挙動の具体例**

例えば「23 + 6」と入力して C を押すと、途中計算表示欄は完全に空欄になり、結果表示欄は「0」に戻ります。

つまり、CE が「部分的なリセット」なのに対し、C は「完全なリセット」なのです。

**まとめ**

クリアキー（C）は、

* CE と異なり「全体を初期化」するキー。
* 内部の数値、表示欄、フラグ、フォント、すべてをリセットする。
* ユーザーが「一度まっさらな状態に戻したい」と思ったときに使う。

これにより、CEとCの役割が明確に分かれ、使い分けができるようになっています。

**7) 桁下げキー（Backspace）**

次にご説明するのは **桁下げキー、いわゆる Backspace キー**です。

このキーの仕様はとてもシンプルで、「一度押すごとに 1 桁ずつ数値を削除する」というものです。

コードの実装は、**行1083から1160の OnBackspaceButton メソッド**に書かれています。

**処理の流れ**

最初に行っているのは、エラー状態かどうかの判定です。  
もしエラー表示中であれば、ShouldResetOnError() が呼ばれ、エラー状態を解除して「0」に戻します。

次に行っているのは、「現在の表示が指数表記かどうか」の判定です。  
指数表記かどうかは IsExponentDisplay() で確認しています。  
もし指数表記であれば、1 桁ずつ削除するのは現実的でないので、強制的に「0」に戻して再入力を待つようにしています。

通常表示の場合は、内部で保持している **m\_lastUserTypedRaw** の末尾を 1 文字削除します。  
そして削除後の文字列を textResult.Text に反映し、さらにカンマ区切りも再計算します。

**具体例**

例1：「1234」と入力されているときに Backspace を 1 回押す。  
結果表示欄は「123」となります。

例2：「0.567」と入力されているときに Backspace を 2 回押す。  
最初に「0.56」、次に「0.5」となります。

例3：指数表記の「2.3e+10」が表示されているときに Backspace を押す。  
この場合は「0」にリセットされ、再入力待ちに戻ります。

**まとめ**

桁下げキーは：

* 通常は 1 桁ずつ削除。
* 指数表記なら強制的にリセット。
* エラー解除にも使える。

という役割を担っています。

**8) 数字キー**

次にご説明するのは **数字キー**です。

仕様としては「0〜9までの数字を入力できる。ただし、最初に 0 を複数回入力した場合は無効。小数点と組み合わせた場合のみ 0.123 のように入力可能」となっています。

コードは、**行487から532の OnDigitButton メソッド**が中心です。

**処理の流れ**

* まず IsInputValid() を呼び出して入力の妥当性を確認します。
* ここで制御しているのが桁数の制限です。
  + MAX\_DISPLAY\_DIGITS = 16
  + MAX\_FRACTIONAL\_DIGITS = 17  
    通常の整数や小数は 16 桁まで。先頭が「0.」の場合のみ 17 桁まで許されています。
* また、同じく IsInputValid() の中で「0 が先頭にある場合の入力制御」も行っています。  
  つまり「00」「0001」といった無効な入力は弾かれます。

その後、m\_textOverwrite の状態に応じて処理が分岐します。

* true の場合 → StartNewNumber() が呼ばれ、新しい入力として扱われる。
* false の場合 → AppendDigit() が呼ばれ、既存の末尾に数字が追加される。

**具体例**

* 「0123」と入力しようとすると、「0」のあとに「1」を押した時点で「1」となり、先頭の 0 は無視されます。
* 「0.123」のように小数点を含めた場合は正しく入力できます。
* 「12345678901234567」と 17 桁目を入力しようとすると無効になります。

**まとめ**

数字キーは：

* 桁数制限を守りつつ入力を反映。
* 先頭 0 の重複は防止。
* 小数点がある場合は 17 桁まで許容。

というルールで実装されています。

**9) 計算命令キー（+ − × ÷）**

次にご説明するのは **計算命令キー**、つまり四則演算のキーです。

仕様では「演算子を押すと途中計算表示欄に演算子が表示され、その時点までの計算が実行される。演算子を連続して押した場合は最後のものが有効になる」となっています。

コードの中心は **行580から626 の OnOperatorButton メソッド**です。

**処理の流れ**

このメソッドでは以下のようなケース分けが行われています。

1. **CE直後の演算子入力** → HandleClearEntryThenOperator
2. **演算子の置き換え** → TryReplaceTrailingOperator
3. **イコール直後の演算子入力** → StartNewChainAfterEqual
4. **%直後の演算子入力** → HandlePercentThenOperator
5. **右辺が未入力のときに演算子変更** → ChangeOperatorWhenRhsMissing
6. **通常の演算子入力** → ComputeThenSetNewOperator

このように、かなり丁寧に場合分けされています。

**具体例**

* 「53 + 123 −」と入力した場合、途中で「53 + 123 = 176」と計算され、式は「176 −」となります。
* 「23 + −」のように演算子を続けて押すと、最後の「−」だけが残ります。

また、除算の場合には **行1371から1381 の Divide メソッド**で 0 除算をチェックしています。  
もし分母が 0 なら、エラーメッセージを表示して処理を中断します。

**まとめ**

計算命令キーは：

* 押された瞬間に中間計算を実行。
* 連続入力時は最後の演算子だけ有効。
* 0 除算は専用処理でエラー化。

という仕組みになっています。

**10) サインチェンジキー（±）**

次にご説明するのは **サインチェンジキー、±** です。

仕様では「入力中の数値の符号を反転する。計算結果に対して押すと negate(...) という形式で途中式に表示される」となっています。

コードは **行1189から1235 の OnToggleSignButton メソッド**と、**行1166から1183 の UpdateExpressionForToggleSign メソッド**です。

**処理の流れ**

* 入力途中の場合 → -value として単純に符号を反転。
* 結果表示中の場合 → negate(値) という形式で途中計算欄に表示。
* 演算子直後の場合 → 右辺に自動的に -A が入力された扱いになる。

**具体例**

* 「100」と入力して ± → 「-100」に変化。
* さらに ± → 「100」に戻る。
* 「100 × 500 =」のあとに ± → 結果は「-50000」、途中計算欄は「negate(50000)」と表示。

このとき、右辺が自動的に生成された場合は m\_lockRhsAfterAutoOp フラグが立ち、編集できないようになっています。

**まとめ**

サインチェンジキーは：

* 入力中 → 符号を反転。
* 計算結果に対して → negate 表記を使う。
* 特殊ケース → 右辺を自動生成してロック。

という柔軟な仕様を持っています。

**11) 小数点キー（.）**

次にご説明するのは \*\*小数点キー（.）\*\*です。

仕様では「数値の直後に小数点を入力できる。重複して入力することはできない。イコール直後の場合は新しい入力として 0. から始まる」と定められています。

コードは **行541から574の OnDotButton メソッド**です。

**処理の流れ**

* すでに m\_numDot = true の場合は無効化。
* m\_textOverwrite = true の場合は、新しい入力として「0.」をセット。
* そうでない場合は、末尾に「.」を追加する。

**具体例**

* 「4」を入力して「.」を押す → 「4.」
* 続けて「8」を押す → 「4.8」
* 「2 × 3 =」のあとに「.」を押す → 「0.」から新しい入力開始。

**まとめ**

小数点キーは：

* 重複禁止。
* イコール直後は新しい小数入力を開始。
* 入力途中なら単純に末尾追加。

というルールで動いています。

**12) イコールキー（=）**

最後にご説明するのは \*\*イコールキー（=）\*\*です。

仕様では「演算子がある場合は計算を確定して表示。演算子がない場合は数値をそのまま表示。連続で押すと同じ演算を繰り返す」となっています。

コードは **行808から836 の OnEqualsButton メソッド**と、**行1459から1543 の ProcessEqualsLogic メソッド**です。

**処理の流れ**

* 演算子がない場合  
  　入力された数値をそのまま結果として表示し、途中計算欄に「=」を付ける。
* 演算子がある場合  
  　m\_firstValue と m\_secondValue を使って計算を確定。  
  　結果を DisplayNumber() に表示し、途中計算欄は「a op b =」の形式になる。
* 連続でイコールを押した場合  
  　m\_secondValue を再利用し、同じ演算を繰り返す。

**具体例**

* 「123 × 5 =」 → 「615」
* もう一度「=」 → 「3075」
* さらに「=」 → 「15375」

**エラー処理**

* 「5 ÷ 0 =」 → Divide メソッドでチェックし、結果欄に「0で割ることはできません」と表示。
* 値が大きすぎる場合 → OverflowException をキャッチして「計算範囲を超えました」と表示。

**エラー処理全般**

最後に、エラー処理について整理します。

**1) エラー条件**

* 計算可能範囲を超えた場合。
* 0で割った場合。

**2) エラーメッセージ**

* 「計算範囲を超えました」
* 「0で割ることはできません」
* 「結果が定義されていません」

これらはすべて Constants.ErrorMessage に定義されています。

**3) エラー時の動作**

エラーになると、

* 結果欄にメッセージを表示。
* フォントサイズをエラーメッセージ用に変更。
* 一部のキー（演算子や%）を無効化する。

これは **行1960から1963の SetButtonsEnabled** で制御されています。

**4) エラー解除**

解除方法は次のとおりです。

* 数字キーを押す。
* CEキーを押す。
* Cキーを押す。
* Backspaceキーを押す。
* イコールキーを押す。

これらの操作の冒頭で ShouldResetOnError() が呼ばれ、内部状態がリセットされて「0」に戻ります。