# **Demonstrating the Halting Problem in C#**

The Halting Problem plays an important role in the modern understanding of computation. In 1936, Alan Turing famously proved that the Halting Problem is formally ‘undecidable’. It provided a concrete example of the more general theorems of incompleteness published by Kurt Gödel in 1931.

This document provides a brief definition of various terms including formal systems, consistency, completeness, effective procedures, decidability, semi-decidability and the meaning of ‘undecidability’. It introduces the Halting Problem and then provides a description of accompanying C# code that demonstrates the Halting Problem in action.

## Formal Systems

Before we can explore the Halting Problem, we must first describe formal systems. A formal system is built on the concept of a formal language. It is a system of symbols, axioms and rules. The following list describes the components of any formal system:

* A finite set of ***symbols*** provides an alphabet. Symbols can be combined to create formulae.
* A set of grammatical 'rules of formation' define how symbols can be combined to create syntactically correct 'well-formed formulae'. These rules are recursive, allowing formulae to be combined to create more complex formulae.
* ***Axioms*** provide a foundational set of propositions that are taken to be true with no further need of proof within the formal system. They provide the foundation for all subsequent reasoning within the system, and often represent statements that are regarded as ‘self-evidently’ true. For example, **Φ ∨ ¬Φ** is an axiom in propositional logic called the 'law of the excluded middle'. It states that a proposition is either true or it is not true.
* ***Rules of inference*** determine how to derive conclusions from grammatically correct premises. These rules are also recursive, allowing final conclusions to be derived from intermediate conclusions.

A formal system is described as ‘recursive’ if there exists some 'effective procedure' to determine whether or not any arbitrary formula is either a valid (well-formed) axiom or a valid conclusion derived by correctly applying rules of inference. An effective procedure is an algorithm that can be followed mechanistically or by rote.

## Decidability

The Halting Problem is a formally undecidable problem. The decidability of any problem is a characteristic of the effective procedure used to determine a solution to that problem. Determination can yield either positive or negative results.

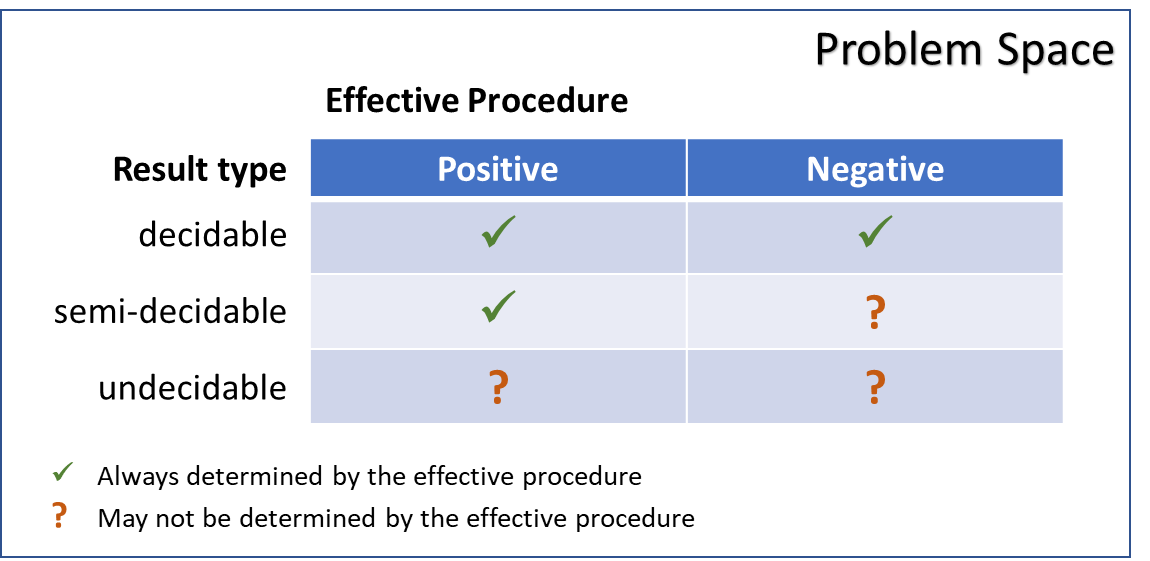
Consider an effective procedure for determining if a given natural number is a member of a finite set of natural numbers. The effective procedure may report a positive result ('yes, it is a member') or a negative result ('no, it is not a member'). If the effective procedure always determines either a positive or negative result, regardless of the natural number we pass to it, we say the problem is 'decidable'.

Some problems are 'semi-decidable'. This means that the effective procedure will only determine positive results, but not negative results. Consider some mechanism for continuously generating random natural numbers of arbitrary length. We can implement an effective procedure for testing if some arbitrary natural number is included in the set of numbers produced by our generator. The effective procedure will continue searching for the given number until it finds it, and then report a positive result. However, the given number may never be generated, so the effective procedure may continue searching forever. It may never determine a negative result because it does not know if the number will ever be generated.

All recursive formal systems support semi-decidability. Any formal system that supports semi-decidability is described as ‘recursively enumerable’. For semi-decidability, it is sufficient for an effective procedure to halt (complete its work in a finite number of steps) only if a formula is determined to be valid within the given system. If the effective procedure establishes invalidity, or fails to establish validity, it may run forever (i.e., never arrive at a conclusion).

Problems can be formally undecidable. In this case, the effective procedure cannot always, and may never, determine a result to the problem. This applies to both positive and negative results. We assume that the effective procedure is correct and acts only on well-formed formulae, and that the formal system in which it operates is logically consistent. A formally undecidable problem has no effective procedure that can always determine a result.

The diagram below summarises the three decidability classes:



If no undecidable problems can be expressed in a logically consistent formal system, we say that the system is 'complete'. However, if any undecidable problems can be expressed, we say that the formal system is 'incomplete'. In the early 20th century, it was not clear if formal systems are always complete or if they can sometimes be incomplete. One famous mathematician, David Hilbert, believed that logically consistent formal systems are always complete. This is equivalent to the claim that any and every correctly formulated mathematical problem has a solution. We now know that he was wrong.

## Gödel’s incompleteness theorems

In September 1930, a young logician called Kurt Gödel briefly described a forthcoming paper, later published in early 1931, at a 'round-table' discussion at a co-located conference in Königsberg (now Kaliningrad). Perhaps just one person at the conference, John von Neumann, properly grasped the implications of Gödel's findings in that paper. By creatively encoding a form of symbolic logical reasoning, Gödel showed that any axiomatic and recursive formal system, capable at least of representing the most basic level of mathematical algebra, cannot be both logically consistent and also complete. This is Gödel's first incompleteness theorem.

The formulae in a logically consistent formal system are devoid of any contradiction. However, contradictions are allowed in an inconsistent formal system. Surprisingly, inconsistent formal systems can play a role in certain areas of mathematical and scientific enquiry, but only as a means of exploring possible solutions in the face of incomplete and inconsistent evidence. Ultimately, we can create effective procedures in an inconsistent formal system to determine both positive and negative results to any instance of a decidable problem expressed within that formal system. Formal systems can either be inconsistent, or incomplete, or both. They can never be consistent and complete.

Gödel realised, rather against his personal expectations, that his work implied a second theorem of incompleteness. It is impossible to determine the consistency of a formal system within the context of that system. This is an undecidable problem, regardless of the consistency of the formal system. This finding helps us to understand an important point. We may still be able to prove the consistency of a formal system. However, we can only do so within the context of a different formal system. Extending this idea, we can also say that, just because a problem is formally undecidable in one formal system, we may still be able to decide the problem in the context of another formal system.

In something of an historical irony, the day after Gödel presented his round-table session, David Hilbert gave the keynote speech at the main conference in Königsberg. He was shortly to retire, and his valedictory speech was a passionate defence of the ultimate decidability of all problems in mathematics and, by extension, as he saw it, in the physical sciences. A decade later, his final words of the speech were carved on his gravestone - "We must know. We will know". Hilbert did not yet know that the logical machinery of problem determination is fundamentally incomplete, and when he did learn this, he made some effort to rethink his position in light of Gödel’s findings.

In 1936, a British logician, Alan Turing, published a paper in which he described a formally undecidable problem in the context of computation. Turing thought of computation in very mechanistic terms. He described the idea of an abstract computational machine (the Turing machine) that can determine the solution to any decidable or semi-decidable problem that can be 'computed'. It soon became clear that his universal computing machine was mathematically equivalent (isomorphic) to existing formal systems, including Alonzo Church's 'lambda calculus' (actually a consistent subset of the original calculus which was shown to be inconsistent) and Emil Post's recursive 'production' system.

Putting this together, we can say that our modern model of computation represents a well-defined and consistent axiomatic and recursive formal system. Gödel’s theorems tell us that this formal system must therefore be incomplete. This, in turn, means that there must be formally undecidable problems in computation. Turing's Halting Problem is one such example.

## The Halting Problem

The Halting Problem can be stated as follows:

“Is it theoretically possible to create an algorithm (an effective procedure) to decide, for every possible computation over a natural number (0, 1, 2, 3, 4, 5… etc.), if that computation never halts?”

Stated in this way, we can see that this problem is framed in accordance with semi-decidability. We are looking for an effective procedure that halts if it can determine, correctly, that a given computation never halts. This procedure can continue forever otherwise. If the effective procedure continues forever, all we can state is that the computation either halts or that it is not possible to determine if it halts or not. We can’t determine which of these possibilities hold true.

Framing a problem in accordance with semi-decidability seems strange to software developers. In programming terms, it means that our code may never halt if it cannot establish that some proposition is true. From a practical viewpoint, the last thing a programmer would normally choose to do is to implement a decision-making procedure that may never halt! However, framing the problem in this way aids our real purpose, which is to illustrate the limits of computability. It reduces the issue to its ‘bare bones’. The Halting Problem is an entirely legitimate problem in computing. If we can find a computation that we know never halts[[1]](#footnote-1), and then show that it is impossible to create an effective procedure that determines this by halting, we will have found a formally undecidable problem that demonstrates a limit to computability.

This is the thrust of Gödel’s incompleteness theorems. When applied to computing systems, they imply that for any logically sound programming language (a language that has the property of being ‘Turing complete’), there must exist decision-based problems that we can legitimately and correctly represent within that language, but which no code written in that language could ever determine. Turing showed that the Halting Problem is one such problem. Given that the language is ‘Turing-complete’, our choice of language is of no consequence here. The result applies to any general programming language.

The Halting Problem provides a concrete example of Gödel’s incompleteness theorems at work. It illustrates the natural limits of computing systems. By ‘concrete’ I don’t necessarily mean that it can be practically demonstrated in an exhaustive fashion. Consider an effective procedure that takes a billion trillion years, and which requires more memory than there are quarks in the universe, to arrive at a conclusion. Writing code for this in C# would a waste of time! In similar vein, how could we be certain that a very long-running procedure will eventually complete, at least in theory, or continue forever?

There are other problems that are insurmountable from a practical perspective. A real-world implementation of an effective procedure for the Halting Problem would never be able to perform a brute-force test of any and every computation over every natural number. There are an infinite number of natural numbers. Turing was researching abstract machines with tapes of unlimited length that could handle an arbitrary number of possible computations in an unlimited length of time[[2]](#footnote-2). We might try to implement an approach based on code analysis, logically reasoning over every code path we find in the given computation that we are testing. Proving that our procedure is exhaustive and correct would be very challenging.

It is not practical to illustrate the Halting Problem here by writing some huge effective procedure to test an infinite number of computations over an infinite number of natural numbers. However, we can show, logically, that there must be at least one or more computations over a natural number whose halting behaviour can never be determined. This is precisely what Turing achieved, several years before anyone created an actual working computer.

Let’s explore the logic. We need to state the notions of computations and the effective procedure formally. Let’s use C to represent a computation and A to represent the algorithm that acts as our effective procedure.

We will consider computations that take a natural number as an argument. We will represent such a number as ‘*n*’. Each computation, then, has the following signature:

C(*n*)

We can provide a unique number to identify each computation. We will use the set of natural numbers to do this, where *q* is the number that identifies each individual computation:

C*q*(*n*)

Let’s now consider A. This algorithmic procedure will search through the problem space. The problem space consists of every possible C*q*(*n*), so A needs to be called repeatedly for every combination of *q* and *n[[3]](#footnote-3)*. A, then, has the following signature:

A(*q*, *n*)

Now we have specified all Cs and A, we can reason thus:

Consider when *q* equals *n*. We substitute *n* for *q*. We can now assert the following:

If A*n*, *n*) halts, then we know that C*n*(*n*) does not halt.

Now consider the following.

A(*n*, *n*) is logically equivalent to A(*n*).

A(*n*) is a computation over a natural number. It is therefore a member of the set of all such computations, i.e., a member of the set of C. We will use *k* to represent the number of this computation. So we can state the following.

A(*n*, *n*) = C*k*(*n*)

We can now re-state the previous assertion for *q* = *n*, as follows:

If C*k*(*n*) halts, then we know that C*n*(*n*) does not halt.

Now consider the situation when *k* equals *n*. In this case, simply substituting *k* for *n* we can re-state the assertion above as follows.

If C*k*(*k*) halts, then we know that C*k*(*k*) does not halt.

This is a contradiction. If C*k*(*k*), which is equal to A(*k*, *k*), were actually to halt, then we would know that it never halts…which it just has! This determination cannot be supported by a logically consistent formal system. Hence, assuming that the logic implemented in our A is sound (which implies logical consistency of the system as a whole), our only option is to infer that C*k*(*k*) never halts. It is the only possible behaviour that C*k*(*k*) could exhibit. It means that our effective procedure, C*k*(*k*), which never halts, cannot determine formally that the computation C*k*(*k*) never halts, even though we know for certain it never does. This feels very un-intuitive. Our effective procedure implements every possible computational method of determining if a computation never halts, so somehow, we have determined something that our algorithmic effective procedure can never determine.

Let’s remind ourselves of precise wording of the Halting Problem:

“Is it theoretically possible to create an algorithm (an effective procedure) to decide, for every possible computation over a natural number (0, 1, 2, 3, 4, 5… etc.), if that computation never halts?”

We have proved that the answer is ‘no’. We characterise the Halting Problem as formally undecidable.

## What does this look like in C#?

We saw earlier that there is no practical way to implement an actual effective procedure that could test every computation over every natural number. The problem space is infinite. However, we could certainly implement code that searches through part of the problem space, and if we did that, we might stumble across an example of a non-halting computation that cannot be determined to be non-halting, even if our effective procedure implements every possible approach for determining if a computation is non-halting.

It turns out that we don’t have to actually implement such a procedure. We can merely simulate the logic. This is very useful. It allows us to simulate the non-halting behaviour of that algorithm. Our demonstration code can therefore determine that it is simulating non-halting behaviour and then conveniently halt and report this. If our algorithm is simulated then the computations it tests can be simulated as well. Our demonstrator only needs to simulate searching through a subset of the real-world problem space and finding a simulated computation that never halts but cannot be determined to never halt.

You may suspect that all this simulation undermines the argument. After all, we could write some code that simply prints a number of simulated results, one of which reports non-halting of our algorithm for a computation we report to be non-halting. Clearly our demonstration code needs to accomplish rather more than that!

In the example code that accompanies this document, we create a subset of the entire problem space as a dictionary of delegates. The index for each dictionary entry represents q. Each computation delegates to a lambda that takes a single unsigned integer, allowing us to test over a subset of all natural numbers.

To simulate the effective procedure, the code implements a method with the following signature:

void DoAssessment(uint computationIndex, uint naturalNumber)

In this method, we test for situations where the two arguments have equal values. In this case, we dispatch to an overload of the DoAssessment method with the following signature:

void DoAssessment(uint naturalNumber)

Both methods do the same thing. They create a string representing A(q,n) and pass this string, together with the computation index and natural number to an internal method that simulates testing the computation for non-halting behaviour. The only difference in the overload is that it specifically tests to see if the indexed computation in the dictionary is the overload method itself. The overload has a signature that is compatible with computation delegates, and the code initialises the dictionary with a number of computations including the DoAssessment overload. If the indexed computation is the DoAssessment method, the string that the method creates represents C*n*(*n*) rather than A(*n*,*n*). This is valid because, of course, A(*n*,*n*) = C*n*(*n*).

The internal test method invokes a helper method that determines if the computation is known not to halt and prints the result using the string created by one of the overloaded DoAssessment methods. The helper method makes its determination in an entirely simulated fashion. By default, the demonstration code uses modulo 3 over the natural number. For each third natural number passed to each computation, the helper method determines that the computation does not halt.

To aid the demonstration, the helper method enters a loop every time it fails to determine that the computation does not halt. This loop is redundant and is included only to represent the notion of semi-decidability. The second time through the loop, the code detects it is in a loop and breaks, returning ‘false’ to represent the semantics of non-determination of the non-halting behaviour of the computation.

The code is written to initially use the default simulated test in all cases. The DoAssessment overload has been carefully added to the computation dictionary at index 6. Hence the simulated test determines that it does not halt and reports the following:

Computation\_6(6) halts, therefore the program knows that Computation\_6(6) does not halt.

This message, of course, makes no logical sense. Our only way to fix what is clearly a logical bug is to add a special test for Computation\_6(6). This test fails to determine if the computation halts. The test is compiled by un-commenting the AssessorTest symbol at the top of the code. Now, when the code runs, the program reports the following:

Computation\_6(6) does not halt, therefore the program does not know if Computation\_6(6) halts.

This message reflects the fact that we know the computation does not halt. The DoAssessment overload, when invoked with the value 6, enters the ‘never-ending’ loop. It is simulating the testing of the DoAssessment overload, as the computation, when invoked with the value 6. The only logical option available to us is to report that the program cannot determine the non-halting behaviour of the computation, even though, as the accessor, the same code reports that it does not halt. The messages deliberately refer to what the program ‘knows’. The reader might wish to reflect on the apparent difference here between the insight available to humans and the knowledge that can be deduced by the programme.

## What does this mean?

We can see that computation, as we understand the concept, has limits. C# is a ‘Turing-Complete’ language running on a practical approximation of a Universal Turing Machine. It is a characteristic of Universal Turing Machines that they are all formally equivalent. Discounting considerations of time (performance) and space (memory), any computational logic that can be executed by one Universal Turing Machine can be executed by all Universal Turing Machines.

Gödel’s incompleteness theorems indicate that there is no way to compute yourself out of the conundrum represented by the Halting Problem. Put simply, there are some computational problems that are not formally decidable through computation, regardless of the sophistication of the machine or, indeed, the programming language.

It is very tempting to make additional deductions of a more philosophical nature. Gödel’s incompleteness theorems have been repeatedly abused in this way, even to the point of supposedly providing ‘proofs’ of both the existence and non-existence of God, based on an entirely fallacious inference from the theorems on the limits of human knowledge. Another controversial, but possibly sound idea is that the incompleteness theorems, and hence the Halting Problem, illustrate some ability of the human mind to attain mathematical insights that are not available via any form of computation, and which are therefore inaccessible to any computer. In the case of the Halting Problem, and based on the proven consistency of the computational model, we somehow seem to know that Ck(k) never halts, even though we know that cannot be computed. The claim that the conscious mind depends on more than mere computation is by no means universally accepted, but the argument has been made cogently and repeatedly. The best-known advocate of this idea is Sir Roger Penrose.

If Penrose and others are correct, it will not prove possible to create a ‘strong’ artificial intelligence that exhibits conscious awareness on the basis, solely, of computation. This is often regarded as equivalent to claiming the impossibility of an artificial general intelligence arising from Turing machines (or rather, their approximations in the real world – e.g., computers). This assumes, of course, that an artificial general intelligence must have some form of conscious awareness – as assertion that is far from certain.

Unfortunately, science has yet to ascertain the mechanisms of consciousness. We do not even know, for example, how general anaesthetics temporarily erase our conscious mind. At this time, it is not possible to prove or disprove any of the numerous claims and conjectures with regard to conscious awareness. We perhaps cannot even determine how a claimed artificial general intelligence, should one arise, could be deemed as having, or not having, ‘true’ consciousness.

Formal undecidability, as encountered in the Halting Problem, is a fundamental, but bewildering, limitation in axiomatic logic, arithmetic and computing. Its true meaning remains a matter of controversy. Its relevance to philosophical enquiry is unclear. Its implications for the future of machine-based reasoning are fiercely debated. I have no answers to these questions, but I hope this small contribution to the subject will help programmers to better understand the nature of incompleteness.

**Charles Young**  
December 2015

**Scooping the Loop Snooper**

an elementary proof of the undecidability of the halting problem

**Geoffrey K. Pullum**, University of Edinburgh

No program can say what another will do.  
Now, I won’t just assert that, I’ll prove it to you:  
I will prove that although you might work ‘til you drop,  
you can’t predict whether a program will stop.

Imagine we have a procedure called P  
that will snoop in the source code of programs to see  
there aren’t infinite loops that go round and around;  
and P prints the word “Fine!” if no looping is found.

You feed in your code, and the input it needs,  
and then P takes them both and it studies and reads  
and computes whether things will all end as they should  
(as opposed to going loopy the way that they could).

Well, the truth is that P cannot possibly be,  
because if you wrote it and gave it to me,  
I could use it to set up a logical bind  
that would shatter your reason and scramble your mind.

Here’s the trick I would use – and it’s simple to do.  
I’d define a procedure – we’ll name the thing Q –  
that would take any program and call P (of course!)  
to tell if it looped, by reading the source;

And if so, Q would simply print “Loop!” and then stop;  
but if no, Q would go right back to the top,  
and start off again, looping endlessly back,  
‘til the universe dies and is frozen and black.

And this program called Q wouldn’t stay on the shelf;  
I would run it, and (fiendishly) feed it itself.  
What behaviour results when I do this with Q?  
When it reads its own source, just what will it do[[4]](#footnote-4)?

If P warns of loops, Q will print “Loop!” and quit;  
yet P is supposed to speak truly of it.  
So if Q’s going to quit, then P should say, “Fine!” –  
which will make Q go back to its very first line!

No matter what P would have done, Q will scoop it:  
Q uses P’s output to make P look stupid.  
If P gets things right then it lies in its tooth;  
and if it speaks falsely, it’s telling the truth!

I’ve created a paradox, neat as can be –  
and simply by using your putative P.  
When you assumed P you stepped into a snare;  
Your assumptions have led you right into my lair.

So, how to escape from this logical mess?  
I don’t have to tell you; I’m sure you can guess.  
By reductio, there cannot possibly be  
a procedure that acts like the mythical P.

You can never discover mechanical means  
for predicting the acts of computing machines.  
It’s something that cannot be done. So we users  
must find our own bugs; our computers are losers!

1. The interesting question here is how to decide a formally undecidable problem. In fact, a problem can only be deemed formally decidable (or undecidable) within the context of a given formal system. If a problem is formally undecidable in one formal system, it may still be formally decidable within another. The Halting Problem describes a formally undecidable problem in computation – that is, no Turing-complete computer can decide the problem. The problem may still be decidable through non-computational methods. [↑](#footnote-ref-1)
2. However, recall that computation may never be completed for a semi-decidable problem. The Halting Problem is semi-decidable. [↑](#footnote-ref-2)
3. Of course, there are an infinite number of natural numbers, and an infinite number of computations over natural numbers. In a real-world implementation, we would be forced to limit the search space. [↑](#footnote-ref-3)
4. But Geoffrey, you’re wrong. It’s not quite like that.

   It really is not where the problem is at.

   You feed Q a number you choose to mean ‘Q’,

   Then weep when you notice that P blows your brew. [↑](#footnote-ref-4)