RC 12402 (#55643) 12/18/86
Computer Science 10 pages

# ABYSS Tokens



Dec. 18, 1986

Bill Strohm (BILL at YKTVMH)
Liam Comerford (CMRFRD at YKTVMH)
Steve R. White (SRWHITE at YKTVMH)

IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

IBM Research Report Number RC 12402

RC 12402 (#55643) 12/18/86
Computer Science 10 pages

# ABYSS Tokens

Bill Strohm, Liam Comerford, Steve R. White

IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

**Abstract**

The problem of authentication is examined briefly. A new authentication technology called the *token* is introduced. The token is an inexpensive hardware device which provides a one-time message to a validating processor. This message is in the form of a response to a random query. The token's response can be predicted by the validating processor. Forgery of a valid token (or simulation of its function) is virtually impossible. Observation of a valid token query/response sequence yields insufficient data to respond to a different query. Since the response is a function of both the query and the specific token being tested, observation of many valid sequences is of no use in attempting to simulate its function as each token will yield only one valid response and each token is different. (That is, the correct response for one token will probably not be correct for another token). The query/response transaction involves no cryptography and the token contains neither cryptographic keys nor cryptographic facilities. Application of this new technology to the ABYSS system is discussed as well as the use of tokens in a generic fashion. A schematic implementation of the token is given and discussed, followed by a quantitative analysis of the security of this new technology.

## Introduction

This paper describes the *token*, an important component of ABYSS (A Basic Yorktown Security System). Although the token can be used as a generic authentication agent in many authentication systems, some familiarity with the ABYSS architecture is recommended to enhance the reader's understanding of the concepts contained in this paper. An introduction to the ABYSS system can be found in [Whit86] and [Cina86].

### *The Concept of Authentication*

Authentication is the act of establishing as genuine or valid. In general, authentication is performed by a sending a telling abstract message from one entity to another. The nature of the message depends on the specific case, but is usually something to the effect of "I am who I say I am." The means for conveying this message fall into three general categories: possession of some transferable unique physical object such as a key, possession of some piece of privileged information such as a password, and unique, identifiable biological features such as one's face or fingerprints. These means, which should not be confused with the message they convey, can be considered as constantly under attack by those seeking to fool the receiving entity for their own purposes. A physical item can be counterfeited. Information can be discovered or guessed. Analysis of biological features calls for some degree of latitude and hence has a potential for being hoaxed.

### *An ABYSS Authentication Scheme*

ABYSS employs a novel authentication agent, the token, to address a particular situation. The situation we consider is one where entity #1 wishes to tell entity #2 that it may to proceed with a particular action once. This situation is analogous to a movie theatre ticket salesperson telling the movie theatre doorman that it is okay to admit a particular person to the theatre. The means is a theatre ticket. When the doorman sees the ticket, he infers that the bearer has met the entrance requirements: he has paid the admission price. The doorman then destroys the ticket, precluding its use for a second, unauthorized, admission.

ABYSS uses a similar scheme to permit a protected processor to install the right to execute a piece of software (embodied in a cryptographic key) in its protected memory. The protected processor can then execute the software subject to the terms and conditions of the software's sale. Entity #1 (the software vendor) needs to convey to entity #2 (the protected processor) that authorization has been granted for it to load the right to execute the piece of software in question. The means is a small machine-readable hardware device called a token. Possession of a valid token represents this authorization. Like a theatre ticket, a token is a one-use device, preventing unauthorized installations on other protected processors. ABYSS differs from the theatre analogy in that while a theatre ticket could presumably be forged with a minimal amount of effort, forging a valid token would require either vast technical resources or preposterous luck.

## Token Concepts

### *What is a Token?*

A token is a small hardware device which is roughly analogous to a theatre ticket. It is examined to ascertain whether or not it is valid and is concomitantly invalidated. The invalidation is inherent in the examination process (not unlike measuring an object's combustion point to test if it is really wood), making it a true read-once medium. This property makes it useful for authorizing a transfer of rights to execute, as it ensures that the transfer will be effected only once. Thus the token is both a validator and a counter.

### *Token Validation*

The way that a token is proven valid is straightforward. The token contains data. Some of this data is read and compared with what the token is supposed to contain. If it matches, the token is valid. What makes the token a unique and viable authentication agent is that its architecture effectively prevents its forgery. That is, even if a the query which requests data from the token and the valid

token response are recorded, no amount of skill or computing power will make this information useful in an attempt to correctly answer a second query.

The token validation process consists of a random series of one-bit queries and responses during which a subset of the token's data is requested and revealed. By the end of this process all of the data contained in the token has been purged, rendering it useless for further operations. To simulate this action (i.e. forge a token) one must be prepared to respond correctly to any sequence of queries. This implies that one must know the entire token contents. There are too many possible combinations of token contents to guess at and they cannot be entirely revealed. In general, at most half of any token's data can be examined. This amount is sufficient to establish that the token is valid, but not enough to afford a prospective forger the information required to simulate a valid token.

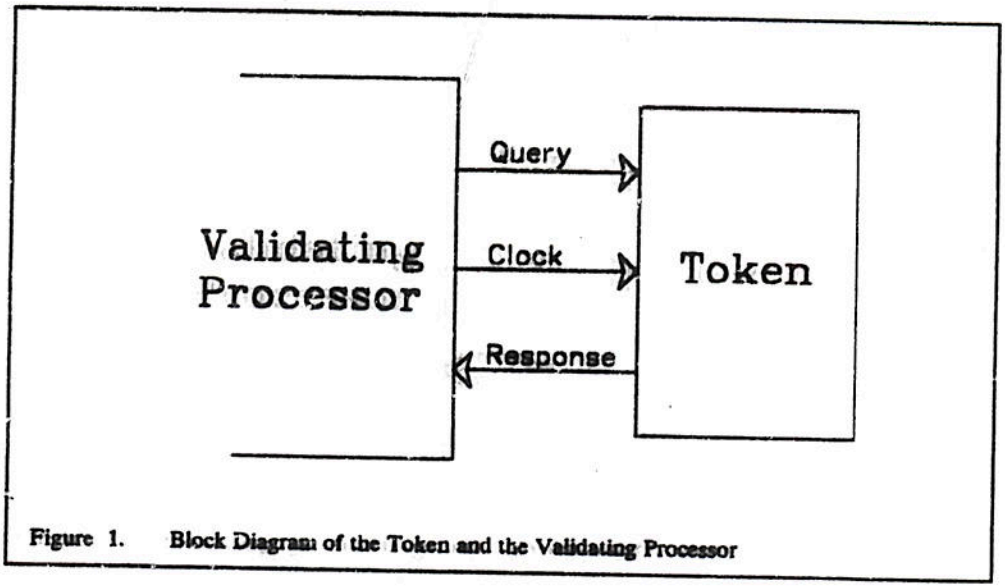


Figure 1. Block Diagram of the Token and the Validating Processor

Figure 1 shows the the logical and physical connection of a token to a validating processor. To test for a valid token, the validating processor generates a random bit sequence, or query. It places the first bit on the token's query line and pulses the token's clock line. The token places the content of the selected register on its output line. The processor reads this value, stores it, and repeats the process with the next bit of the query. After completing the query/response sequence, the processor compares the response sequence with the expected response to the given query. This expected response is found by simulating the token query process using data supplied to the validating processor by other (typically cryptographic) channels. If they match, the token is considered valid.

### *Token Architecture*

The following diagram shows both the simplified token architecture and a sample portion of a token validation sequence. The token's data is stored in two *n*-bit shift registers. These shift registers feed into a multiplexer which routes the data from one of them to the response line.

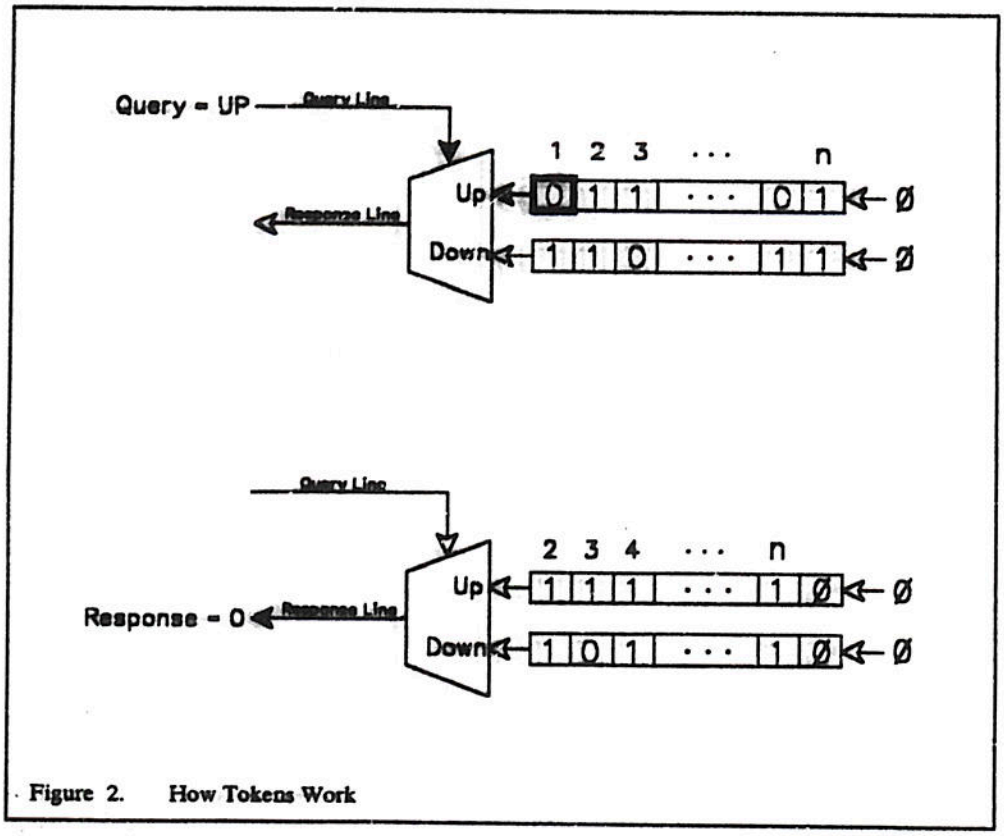


Figure 2. How Tokens Work

In the sample sequence shown, the procedure begins with "what's currently in the UP register?" and the response returned is "zero". The data in the DOWN register went off to never-never land without being seen. This procedure is repeated n times with n randomly selected queries, during which n bits of data are revealed and n bits are discarded.

Each token must contain different data; if all tokens contained the same data, it would be trivial to examine two tokens (read the UP register of one and the DOWN register of the second) and learn the contents of a valid token. A token descriptor containing the token contents must be provided to the validating processor. This token descriptor is encrypted and may be contained within the token itself, in addition to the token contents. The key under which the token descriptor is encrypted is dependent on the application the token is intended to authorize. This allows each token to be fully self-describing, unique, and associated with a specific application.

### *Validation Procedure*

The following pseudo-code describes the process by which the validating processor assesses the validity of a token. Note that the second loop purports to read an encrypted token descriptor from the token. This is the standard way of describing the token's content to the validating processor.

```
TOKEN READ:

query = random[128]                                    /* gen random query     */
index = 0

do 128
    put_query[index]_on_select_line                    /* put query on line    */
    pulse(clock_line)
    read(response_line)                                /* read token response  */
    store_in_token.data                                /* store token response */
    increment_index
    end

do 256
    put_1_on_select_line                               /* set select line      */
    pulse(clock_line)                                  /* read encrypted-      */
    read(response_line)                                /*   token descriptor   */
    store_in_encrypted.token.descriptor                /* and store it...      */
    end

tok.descriptor = decrypt(encrypted.token.descriptor)   /* learn token contents */

expected.data = mask(tok.descriptor,query)             /*select appropriate set*/

flag = compare(expected.data,token.data)               /* pass judgement       */

END

**************************************************************************
  decrypt: Decrypt the given data under a pre-specified key

  mask: Use the query data to select the same 128 bits from the
        token descriptor as were selected during the query/response
        sequence.
**************************************************************************
```

## Token Implementation

The ABYSS implementation of the token chooses a register length of 128 bits. This number affords a very secure system at a small implementation expense. ABYSS tokens also provide a register containing the token's contents in an encrypted form. The ABYSS processor can read this register, decrypt its contents, and know what it should have found during the token interrogation. In the ABYSS system, the key used to encrypt the encrypted token descriptor is the same key used to encrypt the software whose right-to-execute is to be loaded. This ties a token (with random contents) to a particular piece of software.

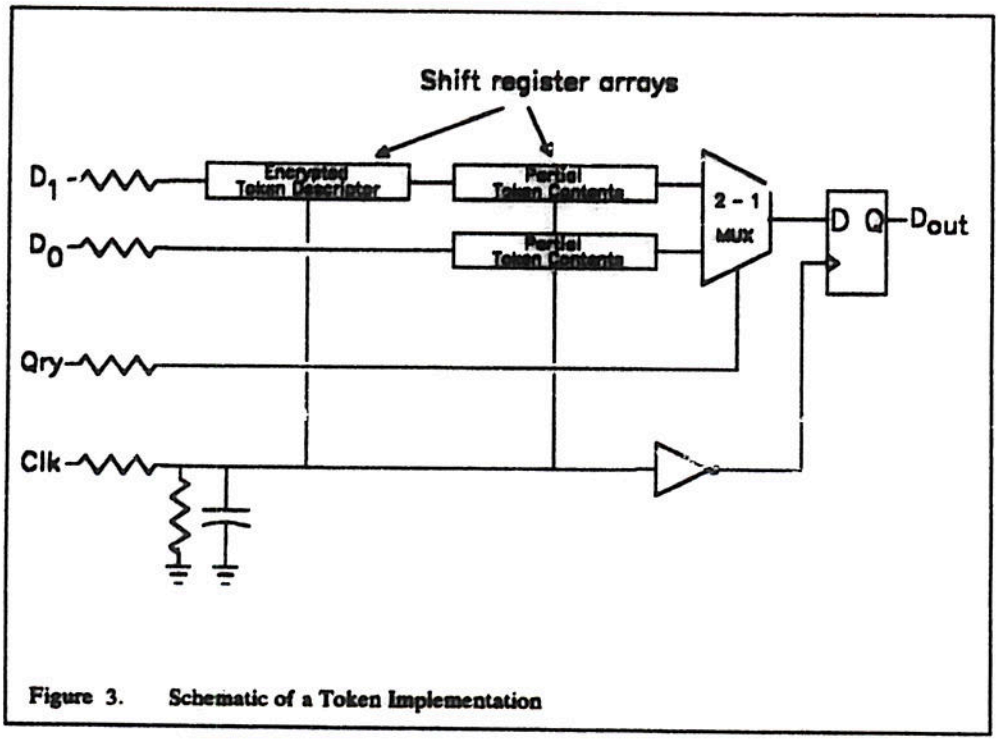


Figure 3. Schematic of a Token Implementation

Figure 3 shows a practical token implementation. Data is loaded into the token using the D0 and D1 lines in parallel. Data is placed on these lines and the clock line pulsed to load the data serially into the shift register arrays. Tokens can be reloaded and reused, given an appropriate set of data.

The preferred token embodiment is as a monolithic CMOS integrated circuit encapsulated in an epoxy package. It is mounted with a calculator type battery on a printed circuit board. The resistors on the input lines are simply protection for the gates to allow the token to be handled freely. The RC network on the clock line input prevents accidental clocking of the shift registers. The $D_{out}$ line is set to zero when the token is loaded, allowing all I/O lines to be grounded with a shorting clip. This further protects the token from static discharge damage or accidental clocking during handling.

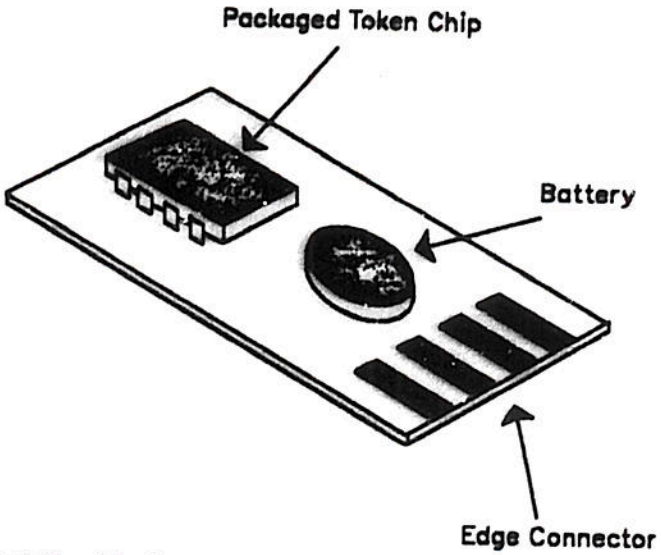


Figure 4. Circuit Board Implementation of a Single-Chip Token

Figure 4 shows a printed circuit board implementation of a token. Future work involves putting tokens into a credit card style package. Future architectural development will aim towards elimination of the battery.

## Token Forgery

### *Overview*

Attempts will undoubtedly be made to gain unauthorized rights to execute software by procuring or simulating valid tokens. Procuring a valid token by means not sanctioned by the software vendor (i.e. stealing a token) will allow one unauthorized copy of the software to be executed. This is a conventional problem and is best dealt with by conventional means (i.e. guarding valid tokens to prevent theft). The more serious attack is simulation of a valid token, which would allow proliferation of unlimited rights to execute the software. Simulation can be attempted repetitively by an attacker attempting to breach ABYSS security. How likely is it that the attacker will succeed?

### *Simulation*

Speaking qualitatively about the nature of what is involved in token simulation, the token contains essentially two 128 bit sets of data, one of which the attacker can learn (by watching a valid token transaction, or by clocking out data with special hardware) and one of which the attacker cannot. The ABYSS processor will select a number of bits from each set for examination. The selection will follow a Gaussian distribution centered about 64 bits selected from each set. The token validation process consists of 128 independent transactions. Each of these transactions consists of the ABYSS processor selecting one of two data bits for examination. The attacker knows the value of one of these bits; if that bit is selected, the attacker can answer correctly. If the other bit is selected, it must be guessed. The attacker has a 50% chance of guessing correctly. This gives a 75% chance of correctly simulating any one transaction. To simulate a valid token, the attacker will have to simulate all 128 transactions correctly. This is hard.

The ABYSS processor simply pronounces the entire process valid or not valid, yielding no information about the accuracy of any single transaction. The forger cannot learn more data bits from a failed simulation attempt.

Quantitatively, the probability of simulating a valid token is:

$$P_{(\text{forged ok})} = \left[P_{(\text{picks known})} + P_{(\text{picks unknown})} \times P_{(\text{guess correct})}\right]^{(\text{total number of transactions})}$$

$$= \left[\frac{1}{2} + \frac{1}{2} \times \frac{1}{2}\right]^{128}$$

$$= \left[\frac{3}{4}\right]^{128}$$

$$= 1.018 \times 10^{-16}$$

It is highly unlikely that the token can be simulated in one try, but what about in $k$ tries? What is the median number of tries required to simulate a valid token? Let $p_{(k)}$ represent the probability that a successful simulation has occurred in $k$ tries. The median for this distribution (i.e. $k$ where the probability of having guessed correctly in $k$ guesses is equal to to the probability of not having guessed correctly in $k$ guesses) can be described in terms of its probability density function as:

$$P_{(k)} = P_{(\text{hit 1st try})} + P_{(\text{hit 2nd try})} \times P_{(\text{missed 1st try})} + \cdots + P_{(\text{hit k-1th try})} \times P_{(\text{missed all previous tries})}$$

$$= p + p \times (1-p) + p \times (1-p)^2 + \cdots + p \times (1-p)^{k-1}$$

$$= p \sum_{i=0}^{k-1} (1-p)^i$$

$$= p\left(\frac{1-(1-p)^k}{1-(1-p)}\right)$$

$$= 1-(1-p)^k$$

For $p_{(k)} = \frac{1}{2}$ (i.e. the median) we can write:

$$\frac{1}{2} = 1-(1-p)^k = (1-p)^k$$

To solve for $k$, we use logarithms:

$$\log_2\left(\frac{1}{2}\right) = k \log_2(1-p)$$

$$k = -\frac{1}{\log_2(1-p)}$$

To facilitate calculation of the denominator of this expression, we use the approximation

$$\log_2(1-\epsilon) = \frac{\ln(1-\epsilon)}{\ln 2} \simeq -\frac{\epsilon}{\ln 2}, \quad \epsilon << 1$$

which yields

$$k \simeq \frac{\ln 2}{p} \simeq \frac{0.693}{p}$$

Inserting $p = 1.018 \times 10^{-16}$ into this equation gives us $k = 6.807 \times 10^{15}$ tries on the average to get a valid simulation. The ABYSS processor controls the rate at which validations may be attempted.

If it allows one try per second, this number of tries will take over 216 million years. This is longer than the projected life of many software products.

Note that this model allows repetition of previous guesses, and thus corresponds more closely to the "ten monkeys at ten typewriters" model of the ABYSS-breaking community. Even for one dedicated forger though, the removal of previously used guesses from the sample space represents a high order term in the overall equation and will not become significant until sometime past the 216 million year mark. It would be prudent, however, to avoid using easily guessed token contents, such as all ones or all zeros.

Once a successful token simulation has been achieved, the forger has typically learned the correct value of 64 more bits. Thus to simulate a token a second time, he must on the average guess only 32 bits correctly. Using the previous model, this will take a median of $6.9 \times 10^7$ tries; about 2.2 years. After the second successful simulation the third comes easier, etc. Eventually, all the data bits will be learned and the token can be forged freely. It is the first simulation that takes a significant amount of effort and makes forgery virtually impossible.

Forgery can be further inhibited by having the ABYSS processor limit the number of attempts to load the right to execute a particular piece of software to some reasonable number. Ten thousand attempts to load a right-to-execute is a definite tip-off that something is not right. Without the ABYSS processor as a willing accomplice (telling the forger whether his guess was correct or not), simulation simply cannot be considered feasible.

### *Other Attack Methods*

The attacker can attempt to decrypt the encrypted token descriptor to learn the token contents. Encryption methods today are sufficiently secure that it is highly improbable that this attack route would be successful. The attacker can attempt to open the token chip and probe the data lines. The data is stored in powered CMOS cells which are corrupted by almost any ambient energy, both from room light and the probes themselves. A layer of glass passivation over the silicon surface makes this procedure almost impossible. Finally, the attacker can try to persuade some dishonest employee of the software vending firm to provide him with the vendor's key or an unprotected version of the software. This is probably the easiest attack method of all.

## Conclusion

The token has been demonstrated to be a viable and secure authentication agent. A valid token is easily detected without the need for esoteric hardware technologies, while retaining its unique forgery-resistant properties. Tokens can be fabricated inexpensively and loaded by a computer or a simple hardware setup.

Rather than being limited to a subsystem of the ABYSS architecture, the token is a generic tool with many applications in other fields - wherever authentication is required.

## References


[Cina86] Vincent Cina Jr., Steve R. White, Liam Comerford, "ABYSS: A Basic Yorktown Security System: PC Software Asset Protection Concepts," IBM Research Report Number RC 12401, Dec. 18, 1986

[Whit86] Steve R. White, Liam Comerford, "ABYSS: A Trusted Architecture for Software Protection," IBM Research Report Number RC 12343, Nov. 24, 1986 (Submitted to 1987 IEEE Symposium on Security and Privacy)

## Appendix A. Token Connection to an IBM PC Printer Port

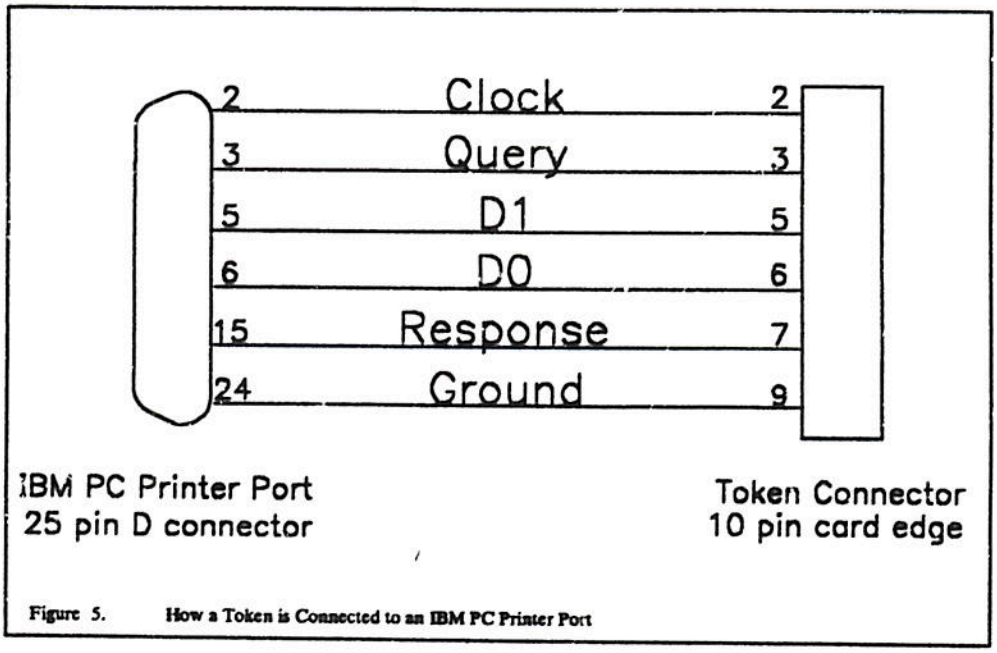


Figure 5. How a Token is Connected to an IBM PC Printer Port