# Universal Getter - uget - Written in Oberon!

**Libraries for**: Gemini, Nex, Spartan, HTTP, HTTPS

---

## Introduction

The goal of this project is to create a universal downloader capable of downloading—and in the future, uploading—files using the protocols: Nex, Spartan, Gemini, HTTP, and HTTPS. This tool is designed to resemble the well-known Unix command-line tool, `wget`, but with added support for a wider variety of protocols.

To better understand the purpose of this project, let's explore the concept of communication protocols.

A communication protocol is a set of rules that defines how data is transmitted between devices or systems. These protocols govern how data is formatted, sent, and received, ensuring effective communication despite differences in hardware, software, or network infrastructures. Some of the most common protocols are HTTP, HTTPS, and TCP/IP (for internet connectivity).

The importance of this project lies in helping individuals who need to download files from less common communication protocols. While accessing and downloading files from HTTPS is easy with various browsers, it is not always the case with less popular protocols. By extending the project into a fully functional Gemini browser, I aim to provide people with the tools to enjoy the same ease of access, whether they're working with HTTPS or other protocols.

---

## What Do We Need for a Universal Downloader?

To develop such a downloader, we first need to understand communication protocols, how they work, and how to handle them effectively. As Michael Mozz, the creator of Spartan, would say: "Let us dive in and explore these topics!" I will do my best to share the knowledge I've gained with the reader.

---

#### Client-Server Model

To establish a connection, we need entities between which the connection can exist. Typically, we don't use a connection without specifying the endpoints involved. This leads us to the concepts of *client* and *server*.

- **Client**: A device or program that requests services or resources from a server.
- **Server**: A device or program that provides services or resources to clients.

Clients initiate communication by sending requests, while servers respond with the requested resources. Together, clients and servers form the backbone of client-server architecture, enabling scalable communication between devices over a network.

In the context of this project, we will need both a client and a server to facilitate downloading (or, as I prefer to call it, "reading bytes"). We decided to use Michael Lazar's Spartan server-client model written in Python, and implement both the server and the client in our preferred language, Oberon. More details on why Oberon is a great choice will be provided in section #####.

---

**Data Structures for Downloading Files**

Next, we need an efficient data structure to read bytes from a server. After researching, we chose a dynamic array. While I didn't know for sure which data structure would be most efficient at first, I spent a week developing a module specifically for dynamic arrays. This module allows us to dynamically allocate space, enabling the reading and writing of bytes from servers.

In the case of HTTPS, we sometimes know the content length in advance, meaning dynamic reallocation isn't always necessary. However, for other protocols, where content length is unknown, dynamic arrays prove useful for handling varying file sizes.

At this point, we are able to read bytes from servers and, eventually, download files.

---

## How Does Downloading Work?

Downloading files involves building requests for each protocol and successfully reading files by dynamically allocating space. For HTTPS, where content length is known, we avoid dynamic reallocation, but for other protocols, like Spartan and Gemini, dynamic arrays are necessary.

All the tools, modules, and implementations developed in this project are open-source and designed to be useful for others. My long-term plan is to extend this project and eventually create a Gemini client browser, providing users with a simple yet effective way to interact with a wide range of protocols.

Let us dive in to the implementation of the dynamic array module.

The project has a `dynamicarray` module that defines the main structure and functions for managing a dynamic array. The array is a flexible data structure that can grow as needed to accommodate more elements.

Here’s the structure:

1. **Types and Variables**
    - `dynamicarray`: This is a pointer to the `DynamicarrayDesc` record, which represents the array.
    - `DynamicarrayDesc`: This record has the following fields:
        - `size`: The current number of elements in the array.
        - `capacity`: The maximum number of elements the array can hold before needing to resize.
        - `content`: A pointer to an array of characters (the dynamic content of the array).
        - `appender`: A procedure for appending data to the array.
2. **Procedures**
    - `Init`: Initializes a dynamic array to a given size with null values.
    - `Create`: Creates and initializes an empty dynamic array.
    - `Resize`: Doubles the capacity of the dynamic array.
    - `Append`: Appends a string to the array. If the array is empty, it allocates memory first; if not, it calls `ReallocAndAppend`.
    - `ReallocAndAppend`: Handles memory reallocation and appends a string to the array.
    - `writetofile`: Writes the content of the array to a file.
    - `copy` and `copylarr0t`: Helper procedures for copying content between arrays.

## Important Code Snippet (Dynamic Array Reallocation and Append)

One of the key functions is `ReallocAndAppendinky`, which is responsible for reallocating memory for the dynamic array when it needs more space to append a new string. Here's the breakdown:

```
PROCEDURE ReallocAndAppendinky(arr: dynamicarray; str: ARRAY OF CHAR);

VAR

  tmp: POINTER TO ARRAY OF CHAR;

BEGIN
```

```
  Out.String("current array content len: "); Out.Int(LEN(arr^.content^), 0); Out.Ln;

  NEW(tmp, LEN(arr^.content^));  (* Create a temporary array with the same size as the original
*)

  Out.String("allocated memory for tmp arr with arr content size"); Out.Ln;


  copy(arr^.content^, tmp^);  (* Copy content of the array to the temporary array *)

  Out.String("copied the arr content into the tmp content"); Out.Ln;


  NEW(arr^.content, LEN(tmp^) + LEN(str) + 1);  (* Allocate more space for the new content *)

  arr^.size := LEN(str);  (* Update the size to the length of the new string *)

  arr^.capacity := arr^.capacity + LEN(str) + 1;  (* Increase the capacity *)

  Out.String("Array size now: "); Out.Int(arr^.size, 0); Out.Ln;

  Out.String("LEN array content size: "); Out.Int(LEN(arr^.content^), 0); Out.Ln;

  Out.String("Array capacity now: "); Out.Int(arr^.capacity, 0); Out.Ln;


  copy(tmp^, arr^.content^);  (* Copy the content back into the original array *)

  Out.String("copied the tmp content back into the array content after reallocation"); Out.Ln;


  copy(str, arr^.content^);  (* Copy the new string into the array content *)

  Out.String("copied the string into the array"); Out.Ln;


  arr^.content^[LEN(tmp^) + LEN(str)] := 0X;  (* Null-terminate the string *)

  Out.String("assigned complete"); Out.Ln;

END ReallocAndAppendinky;
```

## Explanation of the Code

1. **Temporary Array Creation**: A temporary array `tmp` is created to hold the content of the current array. This is done to preserve the existing data before reallocating memory for the new content.
2. **Copy Existing Content**: The content of the original array is copied into the `tmp` array. This ensures that data is not lost during reallocation.
3. **Memory Reallocation**: The memory for the original array is reallocated with enough space to hold both the old content and the new string. The new capacity is updated.
4. **Copy Back the Old Content**: After reallocation, the old content from `tmp` is copied back into the newly allocated memory space.
5. **Append the New String**: The new string is appended to the array, and the array is null-terminated (with `0X`).

This method ensures that the dynamic array can grow dynamically to hold more data as needed.

## Other Key Functions

- **`Append`**: Decides whether to initialize a new array or call `ReallocAndAppend` based on whether the array is currently `NIL` or not.
- **`Init`**: Initializes the array with null values.
- **`writetofile`**: Writes the array content to a file by iterating through the array and writing each character.

Now let us dive in to the implementation details of the Spartan client module. The `SPARTAN` module is a part of a system that provides functionality for creating a simple HTTP client that can connect to a server, send requests, and retrieve responses. It is structured using object-oriented programming (OOP) principles in Oberon, with a focus on low-level socket communication. Here's a breakdown of the key components and functions:

## Data Structures

1. **`transport` and `transportDesc`**: These types represent the transport layer (network communication). The `transport` type is a pointer to a `transportDesc`, which acts as the base type for different transport implementations.
2. **`bsdSocket` and `bsdSocketDesc`**: The `bsdSocket` type extends `transportDesc` and is used to handle BSD-style sockets (network communication) for the client.
3. **`Client` and `ClientDesc`**: The `Client` type is a pointer to a `ClientDesc` record, which represents the client connection to a server. It contains various fields such as:
    - `host`, `port`, and `path`: The target server details.

- `connectionFlag`: A boolean indicating whether the connection was successfully established.
- `rspnBody`: The body of the server's response.
- `header`: The headers of the response.
- `statusCode`: The HTTP status code from the server response.
- `eol` and `null`: Represent end-of-line characters and null characters.
- `dyno`: A dynamic array that stores data.

## Key Procedures

1. **`clearState`**: Resets the state of a `Client` object, clearing response body, headers, status code, and other connection-related information.
2. **`dynomaker`**: Initializes a new dynamic array. This is used to manage response data from the server.
3. **`saver`**: Saves the content from the `dyno` array to a file. This allows the client to persist the response data.
4. **`connector` and `disconnector`**: These procedures handle establishing and closing the connection to the server using the BSD socket.
5. **`writer` and `reader`**: The `writer` sends the HTTP request to the server using the `Internet.Write` function. The `reader` reads the response from the server.
6. **`buildRequest`**: Constructs the HTTP request string based on the `host`, `path`, and other necessary fields. This is sent to the server as part of the GET request.
7. **`parseHeader`**: Parses the response header to extract useful information, such as the status code.
8. **`get`**: This is the main procedure for performing the HTTP GET request. It:
   - Clears the client state.
   - Establishes a connection to the server.
   - Builds and sends the request.
   - Reads the response and stores it in the `rspnBody`.
   - Disconnects from the server.
9. **`Create`**: A constructor for creating a new `Client` object. It initializes the necessary fields (e.g., `host`, `port`, `path`) and assigns the methods like `Connect`, `Disconnect`, `Write`, and `Read`.

## Workflow Example:

1. You create a `Client` instance by calling `Create` with server details (host, port, path).
2. The `get` procedure is called to send an HTTP request and retrieve the server's response.
3. The response data is stored in the `rspnBody` field of the `Client` object.
4. The connection is closed after the operation is complete.

This module is designed to be an efficient and lightweight HTTP client that operates at a low level, directly handling network communication using BSD sockets. It could be expanded further with more advanced features like handling other HTTP methods (POST, PUT, etc.) or managing timeouts and retries. Let us take a look at an important procedure. **`readTillCRLF` (Read until CRLF)**

- **Purpose**: This procedure reads bytes from the socket until it encounters the CRLF (Carriage Return Line Feed) sequence (0x0D 0x0A). It does so byte by byte.

  - The buffer `buf` holds the bytes read.
  - The loop continues reading bytes until it encounters `0x0D` (CR, the first part of CRLF).
  - After reading a `0x0D`, it expects the next byte to be `0x0A` (LF), confirming the CRLF sequence.
  - Once CRLF is detected, the procedure stores the read header into `spartan^.header`.
  - It also calls `parseHeader` to process the header, which typically includes status codes and other metadata.

```
PROCEDURE readTillCRLF(spartan: Client);

VAR

  buf: ARRAY 2 OF CHAR;

  hdr: ARRAY 256 OF CHAR;

  b: BOOLEAN;

  i: INTEGER;

BEGIN

  COPY("", hdr); i := 0;

  REPEAT

        b := Internet.ReadBytes(spartan^.trn(bsdSocket).socket, buf, 1);

        IF b THEN

        IF buf[0] # 0DX THEN

        hdr[i] := buf[0];
```

```
          INC(i);

          END;

          Out.String("have read one byte: '"); Out.Char(buf[0]);

          Out.String("'"); Out.Ln;

          ELSE

          Out.String("failed to read"); Out.Ln; HALT(2);

          END;

  UNTIL buf[0] = 0DX;  (* Until CR (0x0D) is found *)

  hdr[i] := 0X;

  b := Internet.ReadBytes(spartan^.trn(bsdSocket).socket, buf, 1);

  IF b THEN

          IF buf[0] # 0AX THEN  (* Expect LF (0x0A) next *)

          Out.String("0AX expected, got: "); Out.Int(ORD(buf[0]), 0);

          Out.Ln; HALT(1);

          END;

  ELSE

          Out.String("failed to read"); Out.Ln; HALT(3)

  END;

  NEW(spartan^.header, Strings.Length(hdr)+1);

  COPY(hdr, spartan^.header^);

  parseHeader(spartan, spartan^.header^);

END readTillCRLF;
```

- **Process**:
  - The procedure keeps reading bytes until it hits the CRLF sequence.

        ■ When the CRLF is found, the header string is finalized, and further processing takes place (via `parseHeader`).

**`get` (Main Get Request Handling)**

- **Purpose**: This function is responsible for sending an HTTP request and receiving the response. It manages connection setup, request building, and response reading.
- **Explanation**:
    - It first clears any previous state using `clearState`.
    - Establishes a connection using `spartan^.Connect`.
    - Sends an HTTP request using `writer`.
    - Then reads the response using `readTillCRLF` (to handle headers) and `readbuffer` (to handle the body).
    - The response body is stored in `spartan^.rspnBody`.

```
PROCEDURE get*(spartan: Client): strTypes.pstring;

VAR

  req: strTypes.pstring;

  buf: ARRAY 16 OF CHAR;

  i, k: LONGINT;

  b: BOOLEAN;

BEGIN

  spartan^.clearState(spartan);



  spartan^.connectionFlag := spartan.Connect(spartan);

  IF ~spartan^.connectionFlag THEN

        Out.String("Connection failed"); Out.Ln;

        HALT(5);

  END;



  req := buildRequest(spartan);  (* Build the HTTP request *)
```

```
b := writer(spartan, req^);     (* Send the request *)


IF b THEN
        spartan^.dyno := dynomaker(spartan);  (* Create a dynamic array for the response *)
        i := 0; k := 0;
        readTillCRLF(spartan);  (* Read the response header *)
        REPEAT
        zeroBuf(buf);
        b := readbuffer(spartan, buf, i);  (* Read the body in chunks *)
        IF b THEN
        spartan^.dyno.appender(spartan^.dyno, buf);  (* Append body content *)
        ELSE
        Out.String("couldn't connect"); Out.Ln;
        END;
        UNTIL ~b OR (i = 0);
        NEW(spartan^.rspnBody, LEN(spartan^.dyno^.content^));  (* Copy the response body *)
        COPY(spartan^.dyno^.content^, spartan^.rspnBody^);
ELSE
        Out.String("Failed to send request"); Out.Ln;
        HALT(5);
END;


spartan.Disconnect(spartan);  (* Disconnect after the operation *)
RETURN spartan^.rspnBody;
```

END get;

- **Process**:
  - Establishes connection.
  - Sends the request.
  - Reads the response header (using `readTillCRLF`).
  - Reads the body in chunks and appends it to `spartan^.dyno`.
  - After reading the full body, stores it in `spartan^.rspnBody`.

## How It Works

- **Connection Setup**:
  - The `Create` procedure initializes the `Client` object, setting up the host, port, and path, and preparing socket connections.
  - The `Connect` and `Disconnect` procedures handle establishing and tearing down the network connection.
- **Request and Response Handling**:
  - The `get` procedure constructs a request and sends it via `writer`.
  - The `readTillCRLF` function handles reading the headers from the response.
  - The `readbuffer` function reads the body of the response in chunks and stores it in `spartan^.dyno`.
- **Reading and Writing**:
  - `readTillCRLF` ensures that the entire header is read before moving to the body, checking the CRLF sequence for proper formatting.
  - Once the header is processed, the body is read in chunks using `readbuffer`.

This Spartan module is a simple client capable of sending requests and receiving responses over a network socket, implementing basic HTTP-like interactions.

As Gemini module is based off on Spartan module, let me explain their key differences.
The main differences between the *Spartan* module and the *Gemini* module in the Oberon code lie in the functionality and the protocol handling for each. Here's a breakdown:

## 1. Protocol Handling:

- **Spartan Module:** The *Spartan* module is designed for handling the Spartan protocol, though it is not detailed in the code you provided. We can infer that it will likely be structured similarly to the *Gemini* module, with specialized handling for this protocol.
- **Gemini Module:** The *Gemini* module is specifically designed for the Gemini protocol (`gemini://`). It includes functionality to create, connect, send requests, and handle

responses over Gemini, which is a text-based protocol similar to HTTP but designed to be simpler and more focused on security.

## 2. Socket and TLS Handling:

- **Spartan Module:** There is no specific mention of TLS or socket handling for *Spartan* in the code you've provided. We can assume it might use simpler or different methods for managing transport.
- **Gemini Module:** The *Gemini* module makes extensive use of **TLS (Transport Layer Security)** through the `mbedtls` library, which is used for secure communication. It defines a `TLSSocket` type and has procedures like `connect`, `disconnect`, `read`, and `write` specifically designed to manage the secure socket connection. The client also handles entropy, certificates, and personalization data for secure communication.

## 3. Client Structure and Procedures:

- **Spartan Module:** Not explicitly provided in the current code, so it is unclear what client setup is used for the Spartan protocol.
- **Gemini Module:** The *Gemini* module defines a `Client` structure (`ClientDesc`) that holds multiple fields related to the transport layer, certificates, and entropy for TLS. The module also defines a set of procedures for initializing the client (`Init`, `Create`), sending requests (`Write`, `BuildRequest`), receiving responses (`Read`, `Get`), and handling connection states (`Connect`, `Disconnect`).

## 4. Dynamic Arrays and Response Handling:

- **Spartan Module:** While not provided, it may utilize dynamic arrays for handling responses, similar to the Gemini module.
- **Gemini Module:** The *Gemini* module makes use of dynamic arrays (`dynamicarray.dynamicarray`) to store and manage the response body, and it includes procedures for saving, clearing, and appending to these arrays (`clearState`, `dynomaker`, `saver`). The response is read byte by byte and appended to the dynamic array until a complete response is obtained.

## 5. Status and Header Parsing:

- **Spartan Module:** It may have its own mechanisms for parsing responses, but this is not detailed in the provided code.
- **Gemini Module:** The *Gemini* module includes specific procedures like `parseHeader` to parse the response headers, manage status codes, and extract the mimetype. This is critical for processing the response according to the Gemini protocol's rules.

## 6. Personalization and Certificate Management:

- **Spartan Module:** It may have different methods for managing personalization or certificate data, though this is not specified in the provided Spartan code.
- **Gemini Module:** The *Gemini* module includes fields and procedures for handling personalization data (device-specific identifiers) and setting the certificate path (`setcertpath`, `setpers`). It manages these data fields in the `ClientDesc` structure to configure secure TLS connections.

## Certificate Management in the Gemini Module

The module manages certificates primarily for the TLS connection setup, and the code for handling certificates includes functions like setting the certificate path, initializing the client with certificate details, and setting personalization data (like device-specific identifiers).

```
PROCEDURE setcertpath(VAR gemini: Client; path: ARRAY OF CHAR);

BEGIN

  COPY(path, gemini^.crtpath);

END setcertpath;



(* Personalization data (Device specific identifiers) *)

PROCEDURE setpers(VAR gemini: Client; str: ARRAY OF CHAR);

BEGIN

  Out.String("test"); Out.Ln;

  COPY(str, gemini^.pers)

END setpers;



PROCEDURE init(VAR gemini: Client);

BEGIN

  Out.String("entered init"); Out.Ln;

  IF gemini^.trn IS TLSSocket THEN
```

```
        mbedtls.init(gemini^.trn(TLSSocket).netContext, gemini^.trn(TLSSocket).sslContext,
gemini^.trn(TLSSocket).sslConf, gemini^.entropy, gemini^.ctrDrbg, gemini^.cacert, gemini^.pers,
gemini^.crtpath)

  END;

  Out.String("left init"); Out.Ln;

END init;



PROCEDURE connector(VAR gemini: Client):BOOLEAN;

VAR i: LONGINT;

BEGIN

  Out.String("entered connector"); Out.Ln;

  gemini.connectionFlag := FALSE;

  IF gemini^.trn IS TLSSocket THEN

        Out.String("trying to initialize tls socket attrs to the client"); Out.Ln;

        i := mbedtls.connect(gemini^.trn(TLSSocket).netContext,
gemini^.trn(TLSSocket).sslContext, gemini^.trn(TLSSocket).sslConf, gemini(Client)^.entropy,
gemini(Client)^.ctrDrbg, gemini(Client)^.cacert, gemini^.host^, gemini^.port^);



        IF i = 0 THEN gemini^.connectionFlag := TRUE; Out.String("the connection is set to
true"); Out.Ln; ELSE gemini^.connectionFlag := FALSE; Out.String("the connection is set to
false"); Out.Ln; Out.Int(i,0); END;

  ELSE

        Out.String("gemini client is not initialized with tls socket"); Out.Ln;

        HALT(5);

  END;

  RETURN gemini^.connectionFlag

END connector;
```

```
PROCEDURE disconnector(VAR gemini: Client);

VAR

  i: LONGINT;

BEGIN

  IF gemini IS Client THEN

        i := mbedtls.disconnect(gemini^.trn(TLSSocket).netContext,
gemini^.trn(TLSSocket).sslContext, gemini^.trn(TLSSocket).sslConf, gemini(Client)^.entropy,
gemini(Client)^.ctrDrbg);

  ELSE

        Out.String("gemini client is not initialized with tls socket"); Out.Ln;

        HALT(5);

  END;

END disconnector;
```

- **setcertpath**: This procedure takes a path to a certificate file and copies it into the `crtpath` field of the `Client` record.
- **setpers**: This procedure allows setting personalized data, which can include device-specific identifiers. The data is copied into the `pers` field of the `Client`.
- **init**: In the initialization procedure, the `mbedtls.init` function is called to set up the TLS context using the entropy, certificate data, and other details stored in the `Client`.
- **connector**: This procedure sets up a connection, specifically handling the TLS connection initialization using the provided certificate details (like entropy, certificates, etc.).
- **disconnector**: This procedure handles disconnecting the TLS connection and cleans up any resources used.

These procedures allow for configuring the necessary certificates, personalization data, and ensuring secure connections with the server using TLS.

- To summarize, The *Gemini* module is more explicitly defined with rich TLS support and client-server communication over the Gemini protocol. It uses secure connections, parses headers, manages dynamic arrays for responses, and handles certificates and entropy.
- The *Spartan* module is less detailed in the provided code, but it likely serves a similar purpose for the Spartan protocol, with different or simpler transport mechanisms.

The `uget.Mod` module is a program designed to interact with different protocols such as Spartan, Nex, Gemini, HTTP, and HTTPS to retrieve files from remote servers. The program uses command-line arguments to determine the protocol, host, path, and additional options like user authentication for HTTPS connections.

## Key Data Structures and Variables:

- **ARG Record**: Holds details about the URL, including the protocol, host, path, and port.

```
TYPE

 ARG = RECORD

      protocol: ARRAY 64 OF CHAR;

      host: ARRAY 64 OF CHAR;

      path: ARRAY 246 OF CHAR;

      port: ARRAY 5 OF CHAR;

      prothost: strTypes.pstring;

 END;
```

    - **arg**: An instance of the `ARG` record holds the parsed command line arguments for the protocol, host, path, and port.

**Connect Procedures**: These procedures handle the connection to the respective protocol (Spartan, Gemini, Nex, HTTP, HTTPS) and retrieve/save the file.

- **`ConnectSpartan`**, **`ConnectGemini`**, **`ConnectNex`**, **`ConnectHttp`**, **`ConnectHttps`**, and **`ConnectHttpsAuth`** are specialized to handle their respective protocols and support file retrieval.

Example of the `ConnectGemini` procedure:

```
PROCEDURE ConnectGemini;

VAR

  g: GEMINI.Client;

  name, answer: strTypes.pstring;

  a: dynamicarray.dynamicarray;

BEGIN

  g := GEMINI.Create(arg.host, arg.port, arg.path);

  Out.String("trying to initialize socket"); Out.Ln;

  g.Init(g);

  Out.String("left init in connect gemini"); Out.Ln;

  answer := g.Get(g);

  name := getFilename();

  a := g.Save(g, name^);

END ConnectGemini;
```

**`parseCommand`**: Parses the command line arguments into the `ARG` record, identifying the protocol, host, and path, and assigns default ports for each protocol.

**`find`**: Concatenates two strings (`strone` and `strtwo`) with a dot (".") separator and returns the resulting string.

**Command Parsing and Protocol Handling**:

```
PROCEDURE parseCommand(cmd: ARRAY OF CHAR);

VAR
```

```
  i, j: INTEGER;
  host, protocol: ARRAY 64 OF CHAR;
  path: ARRAY 246 OF CHAR;
  port: ARRAY 5 OF CHAR;
  gport, sport, nport, hport, hsport: ARRAY 5 OF CHAR;
BEGIN
  (* Parsing the protocol *)
  WHILE (i < LEN(cmd)) & (cmd[i] # ':') DO
        protocol[i] := cmd[i];
        INC(i);
  END;
  protocol[i] := 0X;

  (* Handling protocol-specific ports *)
  IF protocol = "gemini" THEN
        COPY(gport, port);
  ELSIF protocol = "spartan" THEN
        COPY(sport, port);
  ELSIF protocol = "nex" THEN
        COPY(nport, port);
  ELSIF protocol = "http" THEN
        COPY(hport, port);
  ELSIF protocol = "https" THEN
        COPY(hsport, port);
  ELSE
        Out.String("Unknown protocol: "); Out.String(protocol); Out.Ln;
        HALT(1);
  END;

  (* Parse the host and path *)
  WHILE (i < LEN(cmd)) & (cmd[i] # '/') DO
        host[j] := cmd[i];
        INC(i); INC(j);
  END;
  host[j] := 0X;
  j := 0;

  WHILE (i < LEN(cmd)) & (cmd[i] # 0X) DO
        path[j] := cmd[i];
        INC(i); INC(j);
  END;
  path[j] := 0X;

  arg.host := host;
  arg.protocol := protocol;
  arg.path := path;
  arg.port := port;
END parseCommand;
```

Main procedure logic:

```
PROCEDURE Main;
VAR
  cmd: strTypes.pstring;
BEGIN
  IF Args.argc = 2 THEN
          cmd := getCommand(1);
          parseCommand(cmd^);
          IF arg.protocol = "spartan" THEN
          ConnectSpartan;
          ELSIF arg.protocol = "nex" THEN
          ConnectNex;
          ELSIF arg.protocol = "gemini" THEN
          ConnectGemini;
          ELSIF arg.protocol = "http" THEN
          ConnectHttp;
          ELSIF arg.protocol = "https" THEN
          ConnectHttps;
          ELSE
          Out.String("unknown protocol: '"); Out.String(arg.protocol); Out.Char("'"); Out.Ln;
          END;
  ELSE
          Out.String("Usage: protocol://host/path [-u username -p password]" ); Out.Ln;
  END;

END Main;
```

The `Main` procedure in the `uget.Mod` module is the central logic of the program. It handles the execution flow based on the command-line arguments and invokes the appropriate connection procedure based on the protocol specified by the user. Let's break down its components:

1. **Argument Check (Args.argc = 2)**:
   - The program first checks if there are exactly two command-line arguments (`Args.argc = 2`). This means the user has provided a protocol URL (e.g., `https://example.com/file`).
   - If this condition is met:
     - **`cmd := getCommand(1)`**: The first argument is retrieved (the URL or protocol with path).
     - **`parseCommand(cmd^)`**: The `parseCommand` procedure is called to parse the URL. This extracts the protocol (e.g., `http`, `https`, `gemini`), host, path, and port from the URL.
     - Based on the protocol (e.g., "spartan", "nex", "gemini", "http", "https"), the appropriate connection procedure is called:
       - **`ConnectSpartan`**: For the Spartan protocol.
       - **`ConnectNex`**: For the Nex protocol.
       - **`ConnectGemini`**: For the Gemini protocol.
       - **`ConnectHttp`**: For HTTP protocol.
       - **`ConnectHttps`**: For HTTPS protocol.
     - If the protocol is not recognized, it prints an error message and halts.
2. **Argument Check (Args.argc > 2)**:
   - If there are more than two arguments (`Args.argc > 2`), the program assumes that the user wants to provide authentication details (username and password).
   - The program checks if the second argument is `"-u"`, indicating the start of the username input:
     - **`cmd := getCommand(4)`**: Retrieves the URL or protocol.
     - **`cmd^ = "-p"`**: If the next argument is `"-p"`, it indicates the password follows.
     - The username and password are retrieved using **`usr := getCommand(3)`** and **`pwd := getCommand(5)`**.
     - If the password is empty, a warning message is displayed.

        - After parsing the URL again, if the protocol is HTTPS, it calls **`ConnectHttpsAuth(usr^, pwd^)`** to handle authentication.
        - If the protocol is not HTTPS, it displays a message saying that authentication is not implemented for that protocol.
3. **Error Handling**:
    - If there are fewer than two or more than five arguments, the program displays a usage message and provides guidance on how to format the input.

## Example Usages:

- **Single Argument (No Authentication)**:

uget https://example.com/file.txt

With authentication:
uget https://example.com/file.txt -u username -p password

## Conclusion

In conclusion, the Universal Getter (uget) project aims to provide an easy way to download files from various communication protocols. This tool leverages the power of dynamic arrays for efficient memory management and supports protocols like HTTP, HTTPS, Spartan, and Gemini. The project is still evolving, with plans to expand into a Gemini browser, and I hope it will prove helpful to the wider community.

---

## References

- [Reference 1]
- [Reference 2]

---

## Appendix

- [Appendix details]

---

## Acknowledgments

I would like to especially thank my supervisor, Norayr Chillingaryan, for the immense help with the development and continuous support. I would also like to thank the contributors to the Oberon, Gemini, Spartan communities for their valuable resources and the inspiration behind the project. Thank you.