**Project CodeTalk**

Screen readers are indispensable tools for computer users with visual impairment. Using accessibility features provided by Windows, screen readers interpret and narrate text in standard controls like window titles and textboxes/text areas. However, the need to have generic screen readers that work with any application implies that they are unable to understand semantics of different applications. Consider for example, the Windows narrator reading a page in Microsoft Word. The narrator completely ignores formatting information like headings and titles that are important aspects of document structure. Similarly, in Excel, the narrator does not understand when the user selects a subset of rows in a column and is silent when such a selection is done. This lack of understanding of application semantics ‘hampers the efficacy of standard screen readers substantially. Nowhere is that more obvious than in Visual Studio.

Like all Microsoft applications, VS is written with accessibility in mind and screen readers do interpret the various UI elements that VS offers. However, these readers are oblivious to many aids that Visual Studio offers sighted programmers. The most basic, which is described later in the document, is **glanceability**, i.e. the ability of a programmer to see a file and get contextual information. The equivalent for a visually impaired programmer is to scroll through the file without forgetting what the narrator has read out so far. Imagine doing this for a project of the scale of Windows! Similarly, it is extremely difficult for visually impaired (VI) users to navigate through a code file, or take advantage of intellisense features. The powerful Visual Studio debugger too is complex for a visually impaired programmer as many of its features also require sight. In fact, the VI users we interviewed preferred to use “printf” debugging for the very same reason.

This document is our attempt to enumerate “higher level” accessibility issues with Visual Studio and suggest approaches for fixing the same. In order to better grasp the vast scope of accessibility concerns, we first classify VS usage into the following categories:

* Reading a single code file
* Editing a single code file
* Building a single file (maybe as part of a project)
* Debugging
* Managing projects
* Multi file editing/navigation
* Design features (call graphs, etc.)
* Source control integration
* Project management tools (mostly VSO, this includes tools like Kanban etc.)

Of these, this document focuses primarily on accessibility for single-file reading and editing, as those are the most common user scenarios. We separate the reading and editing scenarios even though reading and editing code often happen simultaneously to highlight the different concerns for the two scenarios. We also briefly examine accessibility concerns associated with debugging, and with the Windows command prompt. For each scenario, we present a list of accessibility concerns and approaches to remedy them. We propose to build a VS plugin (called the CodeTalk) that implements these approaches. We will address the remaining scenarios in a separate document.

## **Reading/Reviewing a single code file**

This user scenario occurs when the user is primarily browsing code written either by themselves or by others. The user opens a single code file either through the command line or by using the File->Open menu, and peruses the file, maybe making minor edits in the process. The following accessibility concerns apply:

### Glanceability

For programmers who can see, a mere glance at the screen provides them with information like:

* The project or file that is currently open
* The current function as determined by cursor placement
* Variables and functions in the file with their scope
* Highlighted text
* Intellisense warnings and errors
* Errors highlighted on the screen

Currently, a visually impaired person cannot take advantage of the glanceability feature of Visual Studio. To remedy this, the CodeTalk plugin will build a summary which can be read out by a screen reader on a keystroke(s). The summarizer will include the following:

1. Name of the current file
2. Types/namespaces/classes in the current file
3. Signature of the current function/class based on cursor position
4. Names of the functions in the file

Another keystroke will be provided to read out the signature of the current function if the cursor is present in the context of a function. The plugin will also provide a separate keystroke to read the “closest” comment; this is particularly useful when the user navigates to the definition of a .NET function [which does not have a function body].

Tree view approach: Another UX option we are considering is to provide a tree view that depicts the hierarchy of classes and members (for OO languages) or the list of functions (for non-OO languages) in an accessible window. This might make it possible for VI users to both identify artifacts in the file quickly and navigate to them with the ease of navigating through an accessible list/tree view. Studies[[1]](#footnote-1) have shown that such a view helps VI programmers complete tasks faster. We plan to conduct additional studies on this approach.

### Accessible intra-file navigation

Sighted users can quickly navigate to the start or end of the block with a single press of the Ctrl + ] key when the cursor is position at the end or the start of the block. Visually impaired users must either mentally keep track of the context of each block or add comments that indicate the context at every block-end statement (as in “end if (a < b)”. The CodeTalk plugin will mitigate this problem by providing a keyboard shortcut that will read out the context of a block end statement (like ending brace). This will be read by a screen reader as:

1. End of function main
2. End of while loop, while (i < n)
3. End of an if condition, if (a == b)

The plugin will also read out the context of the block when the user presses Ctrl + ] to navigate to the start or end of a block. This will offer auditory confirmation to the user.

A keystroke will also be provided to navigate to the start of the current function (for instance, to add a variable declaration). This will be particularly useful as the user will not have to remember the exact function name (as required by search, for instance).

We will also examine the option of providing a command to jump to a specific function by typing the first few characters of the function. Matching functions will be listed and the user will have the choice of navigating to any of them using keyboard strokes.

### Semantic Narration

Current screen readers read the program as text without awareness of language semantics. We plan to make narration more meaningful by adding semantic information to the narration.

For instance,

Instead of reading *int arr[4];* as “int arr open square bracket 4 close square bracket semicolon”

We can read this out as “*integer array of size 4”*.

We are conducting user studies to evaluate this idea in detail.

## **Editing a single file**

Current screen readers offer a competent experience for typing by reading out words as they are typed. We do not propose to make any changes to this behavior at the moment. Some other aspects of editing do however require attention and are listed below.

### File open

Since the command prompt is the most natural “home” for a visually impaired user, they tend to invoke Visual Studio using the “devenv <filename>” command. If the file mentioned does not exist, Visual Studio simply issues a “File not found” message. We plan to offer users a prompt to create the file if the file does not exist. This is similar to the functionality offered by Notepad.

### Flattening command hierarchies

Visual Studio organizes commands into logical hierarchies to make it easier for a sighted user to navigate. An example is the new file dialog which has two list views and hierarchical elements within the left list view. Such dialogs are extremely hard to navigate by the visually impaired user. We propose to flatten most commonly used commands and provide single keystrokes (or a single menu) to access them. Initially, we will provide the New C# file and New C++ file as separate commands. More commands will be added later based on user feedback.

### Error reporting

Visual Studio highlights every syntax and semantic error with a red squiggly even as a user is typing. This allows sighted users to correct code even as they type, keeping the code correct most of the time. This reduces the number of cryptic error messages that the programmer has to wrangle with when the code ­is built. We will make this feature available to visually impaired users by notifying the programmer if any errors are present in the current line. This can be in the form of sound beeps. A user can then use a keyboard shortcut to read out the error details and correct them in place.

### Single file navigation, intellisense and build

Projects are a core concept of working with Visual Studio. Intellisense, navigation (go to definition), build and debugging are enabled only when Visual Studio projects are opened and are not available when a single file is opened. Since VI users (at least the ones we surveyed) find it extremely difficult to navigate the hierarchical, multi-window user interface for managing projects, they prefer to work with single files at a time. This implies that crucial features like intellisense, build and debugging are not available to them.

We propose to provide a useful subset of functionality even when a single file is open. If the file is already part of a project[[2]](#footnote-2), we will offer the following features:

* Navigate to definition of a function/class
* Find all references of a function/class
* Read the documentation comment for the “current” function/class/namespace on a keystroke
* If in a function, read the documentation for the arguments and return types on a keystroke
* Show the “context” of the current line (the closest function/class/namespace that encloses the current line)
* Skip current documentation comment
* Show the DLL where a class/function is defined (even if the DLL has not been added as a reference)
* Autocomplete in the following scenarios:
  + Using/Namespace qualification for types
  + Variable name autocomplete for variables defined in the current file
  + Function name autocomplete for any function in the project
  + Member listing after typing “.” for any class in the project
* Full syntax-checking and basic semantic checks (exact list TBD) even when a single file is open
* We also plan to enable building a single file with “its” project. This requires more investigation as a single file can “belong” to many projects.

If a new file is created, we will automatically add a temporary project that can be saved by the user at a later stage.

*Implementation notes:*

* *A key question is whether the Visual Studio Intellisense database is accessible even when the project is not open. If yes, we can tap into it, but if not, we must implement these features ourselves. We can do this by implementing “CTags[[3]](#footnote-3)” for C# (or any other language we might consider).*

### Preventing accidental edits

A common mistake that VI developers commit is accidently editing an open code file particularly when a block of code is selected. We want to prevent this by (a) warning the user if a chunk of code is selected (with a tone) and (b) identifying accidental edits (for instance, a block of code is selected, but the user presses a keystroke that is typically not used after a selection). We will use a different tone to identify an edit that happened after a selection.

## **Debugging**

VI users face multiple accessibility issues in debugging programs in Visual Studio. Debugging involves changes to the user-interface that are not completely tracked by screen readers. In fact, the Windows narrator completely ignores any changes to the UI interface when the user is debugging a program.

A basic feature the plugin will offer is to give accessibility cues when breakpoints are toggled and when they are hit. In addition, we will offer a new feature we call “way points” that give additional context to the user about the debugging session. Way points are like breakpoints in that they trigger some action when execution reaches a line of code, but unlike breakpoints they can be configured to stop execution or simply continue debugging. Waypoints can be configured to:

* Play a specific tone when execution reaches a line of code. Users can choose different tones to distinguish between different lines of code
* Vocalize a specific message like “Entered while loop”
* Execute expressions and vocalize the result. An example is “for loop entered with loop counter “ + i

We plan to conduct additional user studies to identify features that can further improve the debugging experience for VI programmers.

## **Virtual rotor**

One of the drawbacks of having a rich action set with associated keyboard shortcuts is the threat of running out of keyboard shortcuts. In addition, these shortcuts overload the programmer’s cognitive ability and can quickly become non-intuitive. We are therefore exploring the notion of a “Virtual rotor” to alleviate this problem.

The idea of virtual rotor is to have the Visual Studio environment operate in different modes. By default, Visual Studio will run in the “EDIT” mode in which users can edit read and edit files. Users can cycle through the modes by pressing a particular key combination (say <Up Arrow> + <Down Arrow>). For instance, a function navigation mode can be provided in which, the “n” key navigates to the next function and the “p” key navigates to the previous one. Other modes can be provided for managing projects or for running tools. This design facilitates cleaner and modular interaction while preventing the proliferation of keyboard shortcuts. We intend to conduct A/B tests to test the validity of this approach.

## **Accessible Command Prompt**

The Windows command prompt is an essential tool in the VI programmer’s toolkit. However, it is also completely opaque to screen readers. Screen readers mitigate the issue by offering a “mouse” mode which forces VI users to move the mouse cursor to specific points on the cursor and read text under the cursor. Alternatively, VI users select all the text in the command window and paste it into an editor like notepad to have it read back to them. Thus simple tasks like compiling code or examining the output of a program are unnecessarily ‘verbose’.

We plan to provide a command prompt application that is accessible to screen readers. The precise features of the application are TBD.

## **Screen reader summary**

We provide a quick summary of the benefits and drawbacks of various screen readers both from a user perspective and a “cost-of-implementation” perspective.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature/Implementation Issue** | **JAWS** | **NVDA** | **Window-Eyes** | **Windows Narrator** |
| Usage amongst VI users | Highest | Second highest | Low (must verify) | Low |
| Free/Paid | Payed (40-minute trial available) | Free | Payed (60 day fully functional trial available, 30 minute trials post 30 days) | Free |
| Open source | No | Yes | No | No |
| API Available? | Limited API available in the form of JAWS scripts. | Yes | Yes. (very limited) may be only open to specific partners. | No |
| API Language/interface quality  API Flexibility | JAWS Scripting Language. Limited support to modify speech. | Python. Supports application specific plugins. The API seems to be flexible. Full investigation of its capabilities is still to be done. | API documentation is very outdated. Available documentation shows that the API is very limited in flexibility. | No |
| Ease of VS + reader integration | Easy to integrate with plugin. JAWS’s scripting language does not have the capability to add special sounds, modify speech parameters, etc. We can instruct JAWS to speak specific text to give more context in our case. Exact capabilities and limitations must be verified. | Should be easy to integrate with our plugin. NVDA API allows us to add specific tones on events. This may be helpful to indicate completion of certain tasks or during code editing and debugging. | Integration may be hard due to limited capabilities of the API | Not known |

## **User Studies**

We a want to conduct detailed user studies with two visually impaired users A and B. The following tasks are proposed ranging from easy to difficult:

1. Open an existing file in a specified folder
2. Open a new file
3. Write (and execute) a function to print “Hello World”
4. Write a program that adds and deletes elements from a linked list. The two functions must be separate and invoked from a main program
5. Given an existing file with known source code, list the functions/types/namespaces in the file
6. Given an existing file from an unknown codebase, list the functions/types/namespaces in the file
7. Find a function (go to definition)
8. Given a source file that implements say, binary search, identify the functionality
9. Given a source file with errors, fix the errors
10. More to be added after discussions…

Unless otherwise specified, we assume that the user starts from scratch, i.e. the user is logged in, Visual Studio is NOT running and the user performs the task.

We plan to observe the following data points:

1. Was the task completed?
2. How much time was taken?
3. Was assistance required?
4. What sort of assistance was required?
5. How frequently is the user using the new functionality?
6. Ease of using the proposed keyboard shortcuts.

We also want to conduct the following A/B experiments:

* Using tones for navigation (high tone for start of function, low tone for end of function) instead of verbal commentary
* Testing the efficacy of a virtual rotor mode v/s a dual-mode operation like that of Vim.
* Semantic narration v/s character based narration v/s a mix of both
* Efficiency of navigating using the tree structure of the document v/s navigating using keystrokes (and search) in a flat document

## **Proposed Keyboard shortcuts for CodeTalk**

The keyboard shortcuts proposed use a combination of <CapsLocK> with other keys. The idea is to avoid any overlap with common keyboard shortcuts and have some similarity to readers like Jaws and NVDA. The main key for our plugin (in case we cannot use CapsLock) is still to be decided. This list will not be applicable if we implement the virtual rotor concept.

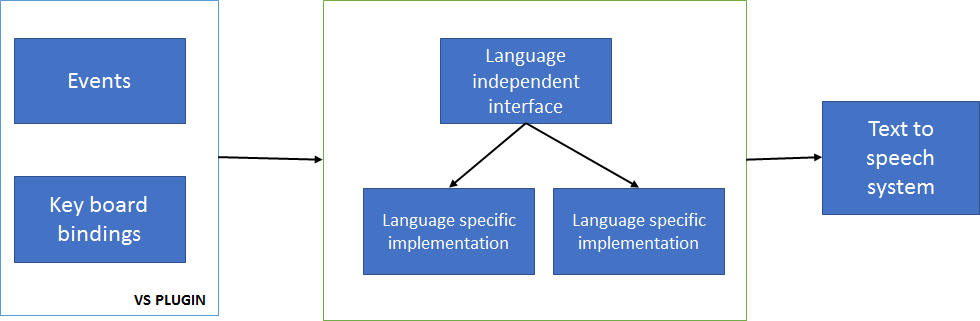
|  |  |  |
| --- | --- | --- |
| Functionality | Description | Keystroke |
| CodeTalk summarizer | Reads the summary of the current code in focus. Helps the user get context. | CapsLock+enter |
| Read function signature | Reads the signature of the function the cursor is placed in. | Alt+CapsLock+s |
| Read closest comment | Reads the closest comment. | CapsLock+alt+c |
| Move to previous function | Moves to previous function. If already in a function body, pressing this keystroke for the first time will move cursor to the beginning of the current function. | CapsLock+Shift+f |
| Move to next function | Moves to the next function. Moves to the current function on first key press. | CapsLock+f |
| Search for function | Gives the user the ability to enter characters from a function name to jump to it. The user should press enter to jump and up or down arrows to move between different suggestions. | Control+CapsLock+f |
| Virtual rotor | Enters a mode where next key press performs a function. More proposed keystrokes based on this to follow. User presses escape to exit this mode. | Up arrow + Down arrow |
| Read mode | Enters read mode. | CapsLock+Space, r |
| Read error details in current line | Reads the error details corresponding to the line in focus. Reports no errors if no errors are present. | Alt+CapsLock+e |
| Announce current indentation | Announces the current level of indentation | Alt+CapsLock+i |
| Move to variable declaration | moves to the current variable’s declaration. | CapsLock+v |
| Search for variable declaration. | Moves to the declaration statement of the variable entered by the user. | Control+CapsLock+v |

## **Implementation**

We will provide the accessibility features by means of a Visual Studio plugin. The initial version will provide the accessibility features described above for **C# programming.** This gives us the advantage of being able to quickly iterate between developing the plugin and getting user feedback. We will use Roslyn for language features (parsing code, identifying functions, types, etc.).

The main blocks in our implementation are as follows:

1. The plugin code with UI: The UI code will invoke commands or events and call an appropriate function in the language specific block
2. Language specific blocks: In our first phase, this will have functions related to C# and Roslyn.
3. Text annotator: This block will annotate a given line of text. For instance, arr[4] will be converted into an “array of size 4”.
4. Speech block: This will invoke a Text to Speech library with the given input. The interface to the TTS system will be through a screen reader. This interface will be implemented through a web service to enable integration with different screen readers like JAWS and NVDA.



We will provide intuitive keyboard shortcuts for all the commands mentioned above. Most commands can be invoked with a <CapsLock> + <Key> combination.

## **Future phases**

In the future phases, we will expand accessibility features to cover the user scenarios mentioned in the beginning of the document. We also plan to provide similar features for VS Code and expand language support to popular languages like Python and Java.

1. http://dl.acm.org/citation.cfm?id=2702589 [↑](#footnote-ref-1)
2. A file can belong to one or many projects, or not be a part of a project at all. We plan to resolve this by assigning the file to any project that is in the same folder as the file, and if no project file exists, walk up the directory hierarchy and obtain all projects that reference the said file, and give the user a choice of “opening” any of the projects. If no such project file is found, we will create a temporary project and assign the file to it. [↑](#footnote-ref-2)
3. http://ctags.sourceforge.net/ [↑](#footnote-ref-3)