JAPANESE INPUT METHOD INDEPENDENT OF APPLICATIONS

By Takahide Honma, Hiroyoshi Baba, and Kuniaki Takizawa

ABSTRACT

The Japanese input method is a complex procedure involving preediting operations. An application that accepts Japanese from an input device must have three systems for the input method: a keybinding system, a manipulator for preediting, and a kana-to-kanji conversion system. Various keybinding systems and manipulators accelerate input operations. Our implementation separates an application from the Japanese input method in three layers. An application can use a front-end input processor to perform all operations including I/O. An application can use the henkan (conversion) module and implement I/O operation itself. An application can execute all operations except keybinding, which is handled by an input method library.

INTRODUCTION

In this paper, we first present an overview of the technical environment of the Japanese input method implementation. Based on this overview, we briefly describe the critical engineering issues for conversion of Digital's products for the Japanese user. Our most critical engineering issue was the reduction of similar (but slightly different) work to localize products. Another issue was to satisfy customers' requests for the ability to use the many input styles familiar to them. We describe our approach to the development of a Japanese input method that overcomes these issues by separating the input method from an application in three layers.

OVERVIEW OF THE JAPANESE INPUT METHOD

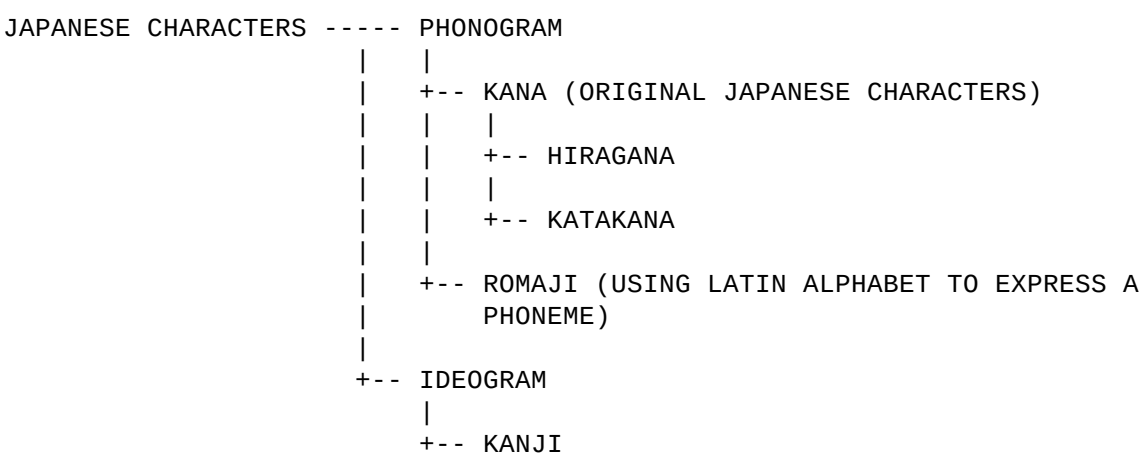
In this section, we describe Japanese input and string manipulation from the perspective of both the user and the application. Based on these descriptions, we present a brief overview of reengineering a product for Japanese users and a summary of the industry's complex techniques developed for Japanese input methods.

Japanese Input

The Japanese writing system uses hundreds of ideograms called kanji. In addition, Japanese uses a phonetic system of kana characters (hiragana and katakana) and has accepted romaji, which is the use of Latin letters to spell Japanese words. Figure 1 summarizes the Japanese character systems. Japanese input requires users to operate in a "preediting" mode

to convert kana or romaji into a kana-kanji string.[1,2]

Figure 1 Japanese Character Systems



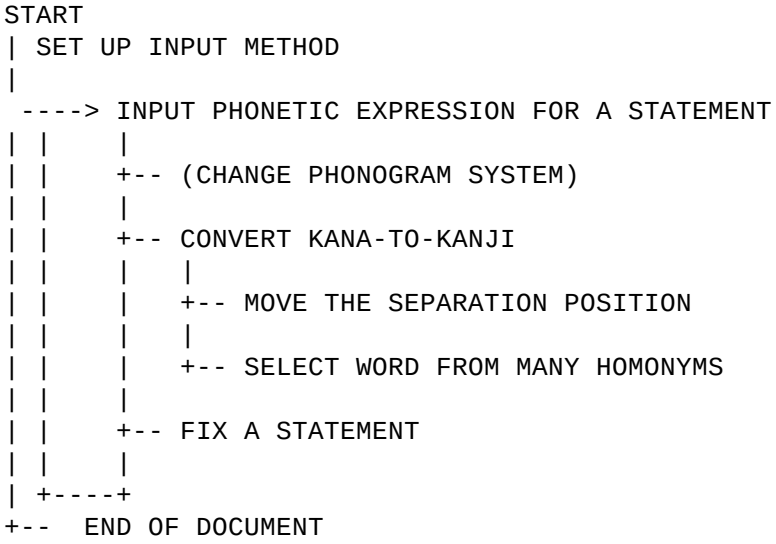
The computer keyboard used for Japanese input has multiple character assignments. Almost all keys on the Japanese keyboard are assigned both a Latin alphabet character and a Japanese kana character. The Japanese user must first choose between kana key input or alphabet input. A user in an engineering area generally uses romaji (alphabet) key input. In the office environment, however, a user prefers kana key input because it requires half as many keystrokes as romaji input.

Preediting Operation

The user inputs the phonetic expression in either kana or romaji that represents the statement the user wants to input. Then the user presses the conversion key to convert the phonetic expression to a kana-kanji mixed string. At this time, the user checks the accuracy of the conversion result. Sometimes the user needs to select the correct word from a system-generated list of several homonyms. Moreover, a user may also need to determine the separation positions in the phonetic expression to ensure a meaningful grammatical construction.

Japanese has no word separator equivalent to the space character in English. To obtain the correct or expected separation of grammatical elements, the user must sometimes move the separation position. After the user constructs a corrected statement, he or she finishes preediting and fixes the statement. The user repeats these complex steps to construct Japanese documents. Figure 2 shows the preediting steps for the Japanese user.

Figure 2 Preediting Japanese Input



Various techniques have been developed to accelerate Japanese input operations. They include UNDO, COPY, zip code conversion, and categorized expert dictionary.

Japanese Application Capabilities

The Japanese application has two special capabilities for Japanese processing. First, the application must be capable of handling multibyte characters. This subject itself is interesting as it involves wchar_t and Unicode character sets; however, this paper focuses on the second capability, the implementation of the input method. An application that accepts Japanese from an input device must have, at least, three additional systems for the input method. These are the so-called keybinding system, a manipulator for preediting, and the kana-to-kanji conversion system.

Keybinding System. This system analyzes the key input from a user and determines which of the key's functions the user wants to do. It defines the user interface and the way a user operates with keystrokes. It also defines the preediting conversion key. We imagine there are as many keybinding systems as there are word processors.

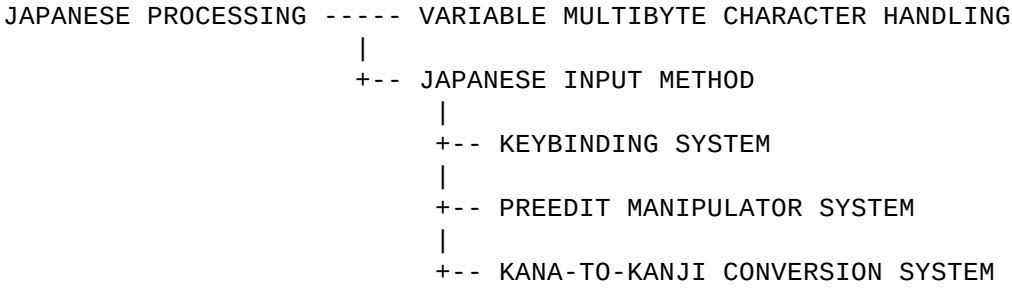
Preedit Manipulator System. This system not only echoes the input characters on the screen but also controls the video attribute that expresses the preediting area. This capability allows the user to distinguish preediting strings from background fixed strings. The user must be able to recognize the preediting string for more processing (for example, to convert the input to another expression such as kana to kanji). In addition, the user

can set this system to convert input to another expression dynamically (for example, automatic conversion of romaji to kana).

Kana-to-kanji Conversion System. This system analyzes the input string, gets the word from a dictionary, and constructs the correct statement grammatically. Many personal computer (PC) vendors have invested in systems that use this input method. In Japan, some vendors have introduced artificial intelligence technology, but this system essentially has only statistical rules.[3,4]

Figure 3 summarizes Japanese processing as handled by applications.

Figure 3 Japanese Processing by Applications



Method of Japanese Conversion

As mentioned above, to convert a product for use in Japanese, we must implement both a Japanese-string manipulator and an input method. To retain the "look-and-feel" of the original Digital product, the interface is designed so the Japanese user does not need to explicitly enter the preediting session with the special-purpose key (Enter Preedit) but is automatically entered. With most applications on other systems, a user must explicitly enter the preediting session by using the special keystroke. This implementation has the advantage that it completely separates the input method from the application, but it requires the user to remember to perform an extra step.

To eliminate the conflict between the original product's key function and the additional Japanese input function, each product has to have a slightly different keybinding system for Japanese. As a result, a user must learn more than one Japanese input operation when using multiple products.

User Environments

PCs are widely used in many offices and are popular devices for

Japanese input. Naturally a user wants to operate with a familiar PC keystroke for Japanese input even in integrated systems (in some servers). When PCs, which use front-end processors, are integrated into environments with VMS and UNIX systems, a user often prefers the PC's interface. The more integrated a user's environment is, the more requirements a user has.

In addition, a distribution kit for the X Window System in a UNIX environment has some sample implementations of the Japanese input method. This kit gives a user more choices for input at no cost.

The market for the Japanese input method separates vendors into two main groups. One is the PC front-end processor manufacturer who implements more advanced techniques but at a high price. The other is the UNIX system vendor who supplies input implementations free (without guarantee) and thus reduces the maintenance cost.

In the next section, we present our approach to the development of an application-independent Japanese input method. The goals of our design were (1) to include the PC keybinding system in integrated environments so users could select their preferred input method, (2) to supply a tool that would easily convert products for the Japanese user, and (3) to provide a way to access the interfaces of several Japanese engines and thus capture the free software capabilities.

APPLICATION-INDEPENDENT APPROACH

As described in the previous section, the Japanese input method includes complex techniques. Many PC software vendors (but not manufacturers) decided against developing their own methods and incorporated a popular input method for their applications. This decision, of course, reduces their development cost. Our approach also seeks to reduce development cost. We separated the input method from the application to the greatest extent possible, as long as the separation did not adversely affect the application.

The PC system is designed as a single-task system, but Digital's operating systems (OpenVMS, ULTRIX, and DEC OSF/1 AXP) are designed as multitasking systems. Therefore we could not adopt many of the PC techniques that were implemented in the driver level. For example, access to dictionary and grammatical analysis of the input string are too expensive in the driver level of a multitasking system because they use system resources that are common to all tasks on the system.

Our approach divides the input method into three layers. Each layer is dependent on any lower layer. Consequently, any application using the highest layer also uses the functionalities of the other two layers.

Strategy of Three Layers

The criteria of our layering strategy were (1) to reduce the cost of reengineering products for the Japanese user, (2) to unify the input method user interface, and (3) to protect the user's operational investment in a keybinding system.

These criteria led us to set the keybinding system into the lowest layer. We named our system the input method library (IMLIB) and released it on VMS/Japanese version 5.5 and ULTRIX/Japanese version 4.3. We also ported it to the Alpha AXP system, and this facility is available on any Japanese platform. Any application using our method needs to use IMLIB.

In essence, this keybinding system allows a user to change the input method of operation to any style by changing the IMLIB definition files. If an application supports IMLIB, a user can change the application's input operation by changing IMLIB once. As a result, an application's key customization function can move into IMLIB.

At this point, we considered the simplest method of separating the input method from applications. The intermediate process, also called the front-end method, processes all the input and then passes it to an application. Many front-end implementations use the pseudo-terminal driver (pty in UNIX or FT in OpenVMS). The intermediate process gets all I/O to and from an application, processes it, and finally passes it to an application or a device. This implementation cannot recognize the application input requests, but works only by a user's operation. To change this operation, we set the hook inside the terminal driver to get all application-request information. Our front-end process recognizes application requests and works without conflict.

One advantage of this front-end implementation is a complete independence of applications. This can also be a disadvantage since an application cannot control the input method closely. For example, this implementation can alter the user interface of an editor system.

We continued to study another layer for separation. The preediting operation, that is, all the input string manipulations except I/O to devices, was a candidate. All applications pass the input from input devices into the Japanese input manipulator and then pass the output from this manipulator onto output devices. By using this system, we can unify the input operation except for device-dependent operations and reduce the cost to implement this kind of functionality.

Our development process started at the lowest layer (IMLIB), proceeded to the highest layer (front-end), and finished at the middle layer (preediting manipulator). In the following sections, we describe the functionalities in each layer from the lowest to the highest layer.

IMPLEMENTATION OF IMLIB

IMLIB is a utility that supplies the keybinding definition function and other information for customizing the Japanese input operation. This capability enables us to supply user-friendly keybinding systems. A user can change the input sequences and the look-and-feel of the user interface by modifying databases. We introduced two databases, KEYBIND for keybinding and PROFILE for look-and-feel and an application's usage. We also supplied the KEYBIND compiler for improved performance and the elimination of the grammatical error at run time.

As mentioned in the introduction, there are many implementations of Japanese input styles on PCs or some word processors, and some text editors on various operating systems. If a user needs to use a different editor, he or she needs to learn another operation. Our method unifies the input operation. We studied several types of input styles and recognized that we could build the general model for this input operation. The IMLIB manual describes this model in detail.[5,6] In this paper, we discuss it briefly.

KEYBIND Database

In the Japanese input operation, entering the key input causes several conversion actions and state transitions. Figure 4 shows the multiple transitions incurred during input. We needed to define the conversion actions and some state transitions as a single key input action. We implemented this function through the KEYBIND database and language. Figure 5 is an example of the KEYBIND database. A user builds an input style by changing the KEYBIND database with the KEYBIND language.

Figure 4 State Transition

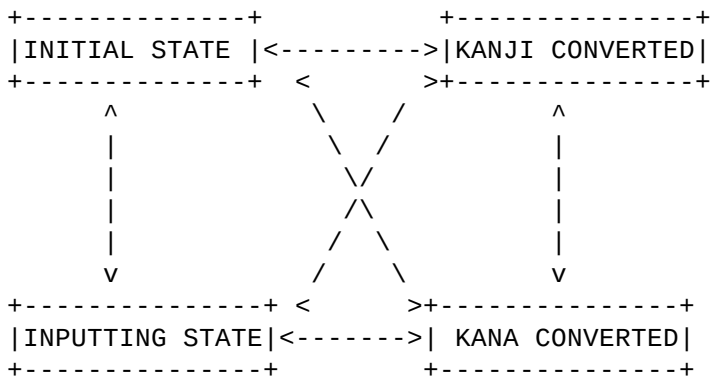


Figure 5 Portion of KEYBIND Database

```
! (JVMS conversion key definition file (system template) ver 1.0)
!
    gold            = CTRL_G;               ! Gold key; used as a PREFIX key
    kakutei         = CTRL_N;               ! Finish without any conversion
    kanji_henkan    = NULL, gold + CTRL_K;  ! Convert to Kanji / next
candidate
    hiragana_henkan = CTRL_L;               ! (Convert clause) to Hiragana
    katakana_henkan = CTRL_K;               ! (Convert clause) to Katakana
    zenkaku_henkan  = CTRL_F;               ! Convert to full width characters
    hankaku_henkan  = gold + CTRL_F;        ! Convert to half width characters
    kigou_henkan    = GS;                   ! Symbolic code conversion
    oomoji          = VOID;                 ! Convert to upper characters
    komoji          = VOID;                 ! Convert to lower characters
    ji_bunsetsu     = CTRL_P;               ! Move to next clause
    zen_bunsetsu    = gold + CTRL_P;        ! Move to previous clause
    tansyuku        = US;                   ! Shrink the clause
    sintyou         = gold + US;            ! Extend the clause
    zen_kouho       = gold + (NULL, CTRL_L);! Previous clause candidate
    kaijo           = CTRL_N;               ! Cancel the conversion
                                            !   and go into input state
    sakujo          = DEL;                  ! Delete previous character
    hidari          = LEFT;                 ! Move the cursor left
    migi            = RIGHT;                ! Move the cursor right
    space_first     = "\ ";                 ! Finish by space
                                            !  (space at initial state)
    space_input     = "\ ";                 ! Finish by space
                                            !  (space at other states)

STATE "initial" =
    space_first           : NONE;
    kanji_henkan          : START_SELECTED, CONVERT, GOTO "kk_converting";
    hiragana_henkan       : START_SELECTED, HIRAGANA, GOTO "converting";
    katakana_henkan       : START_SELECTED, KATAKANA, GOTO "converting";
    zenkaku_henkan        : START_SELECTED, ZENKAKU, GOTO "converting";
    hankaku_henkan        : START_SELECTED, HANKAKU, GOTO "converting";
    kigou_henkan          : START_SELECTED, SYMBOL, GOTO "converting";
    oomoji                : START_SELECTED, UPPER, GOTO "converting";
    komoji                : START_SELECTED, LOWER, GOTO "converting";
    TYPING_KEYS           : START, ECHO, GOTO "inputting";
    END;
          ...
```

IMLIB allows the user to change the keybinding and to choose a different input sequence with a different state transition vector. For the user's convenience, IMLIB provides some KEYBIND databases of the major Japanese input styles in default.

When an application calls the ImSetKeybind function, it loads a KEYBIND binary file into memory. Each time the application gets the key input, it queries the key's function from IMLIB. IMLIB searches the KEYBIND file for the key's definition and returns that information, called an action, to the application. Each

action is a set of orders that has a different procedure for Japanese conversion. For example, the action CONVERT means to convert an input string to a kanji string. At that time, IMLIB also maintains Japanese input states and, if necessary, changes the state.

PROFILE Database

The Japanese input operation has many parameters to determine its look-and-feel, such as the video attribute for the preediting string, preediting exception handling, and application-specific processing. The PROFILE database stores these additional parameters the same way as the resource file does in the X Window System.

The PROFILE database is a text file. It contains several records that represent each environment. This record format has the style of INDEX : value. The application predefines the INDEX for its purpose; however, IMLIB defines some INDEXes related to Japanese input operation because it requires some common environment definitions. The range or value corresponding to the INDEX is placed in the right-hand side of the record. Figure 6 shows a record from a PROFILE database.

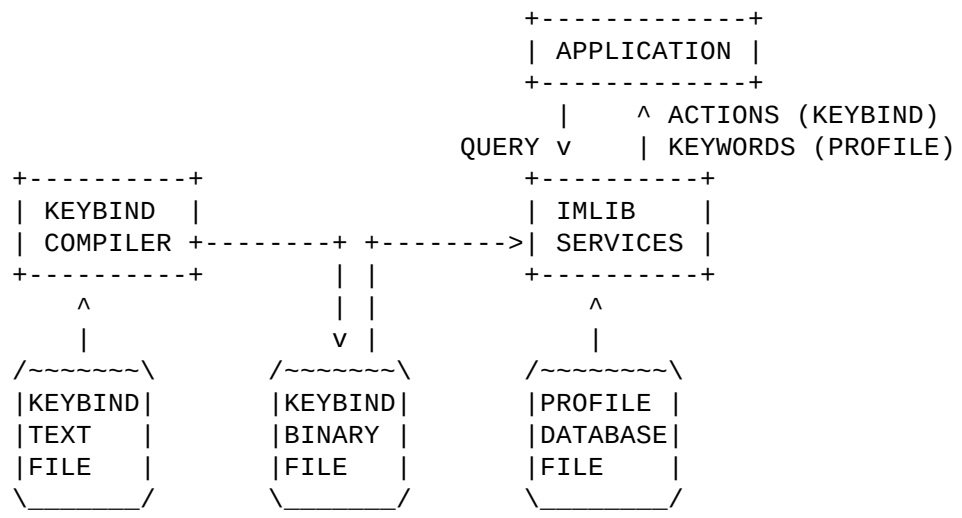
Figure 6 PROFILE Database Record

```
DEC-JAPANESE.KEY.keybind : im_key_jvms_level2
DEC-JAPANESE.KEY.keybind_1 : im_key_jvms
DEC-JAPANESE.DISP.preEditRow : current
DEC-JAPANESE.DISP.preEditColumn : current
DEC-JAPANESE.DISP.inputRendition : bold
DEC-JAPANESE.DISP.kanaRendition : bold
DEC-JAPANESE.DISP.currentClauseRendition : reverse
DEC-JAPANESE.DISP.leadingClauseRendition : none
DEC-JAPANESE.DISP.trailingClauseRendition : none
DEC-JAPANESE.ECHO.ascii : hankaku
DEC-JAPANESE.ECHO.kana : hiragana
DEC-JAPANESE.ECHO.autoRomanKana : off
DEC-JAPANESE.OUTRANGE.clauseSize : none
DEC-JAPANESE.OUTRANGE.clauseNumber : rotate
DEC-JAPANESE.OUTRANGE.cursorPosition : done
```

KEYBIND Compiler

The KEYBIND compiler analyzes the KEYBIND text file and creates the KEYBIND binary file. IMLIB services reads the PROFILE database and the KEYBIND binary file and maintains them in memory. As a response to an application's query, IMLIB services sends it the actions in KEYBIND and the data in PROFILE and at this time maintains the KEYBIND states. Figure 7 shows the relationship among the IMLIB components.

Figure 7 Relationship among IMLIB Components



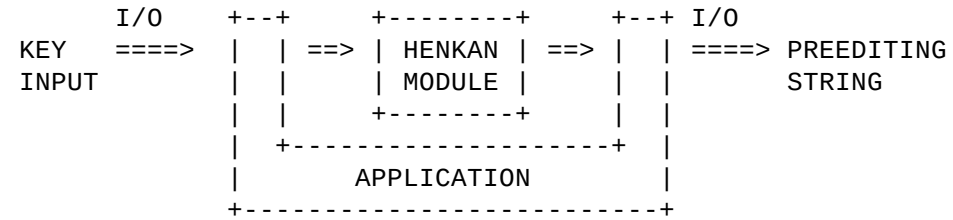
IMLIB is available on the OpenVMS VAX, OpenVMS AXP, ULTRIX, and DEC OSF/1 AXP operating systems. The major applications are DECwindows/Motif, DECwrite, the front-end input process, and screen management (SMG).

IMPLEMENTATION OF THE HENKAN MODULE

The second layer is part of the Japanese input manipulator and is called the henkan module or HM. (Henkan means conversion in Japanese.) It does not handle I/O operation but accepts key input from the application and converts it to a string in preediting.

Figure 8 summarizes the function of HM. An application passes the key input to HM stroke by stroke. HM performs all Japanese preediting operations; the application has no direct manipulation of the input. Then the application gets the preediting string from HM. Because HM does no I/O, it is independent of any specific device. As a result, all applications, including windowing systems, can use HM.

Figure 8 Henkan Module Function



In addition, HM can access the resources of IMLIB. This feature

helps the unification of the Japanese input user interface and reduces the number of similar product conversions. HM has another significant capability. We defined the common (minimum) application programming interface to potentially accept all Japanese conversion engines and implemented "PLUGGS" in HM. Therefore HM can use one or more engines for kana-to-kanji conversion.

HM Mechanism Overview

HM is a tool that any application can use. An application passes key input to HM by a normal procedure call. After HM processes it, HM calls application routines with the processed result. Because HM handles large string buffers, it dynamically allocates/deallocates memory. To ensure that memory is retained, we used a callback technique. (These techniques are described later in the Callback Routines section.)

HM operates by key input as follows:

1. HM gets a keycode from an application with procedure arguments.

2. HM gets the actions assigned to the key from IMLIB.

3. If the key is not assigned to the Japanese input operation, HM tells the application to process it separately.

4. If the key is assigned to the Japanese input operation, HM processes it according to the actions.

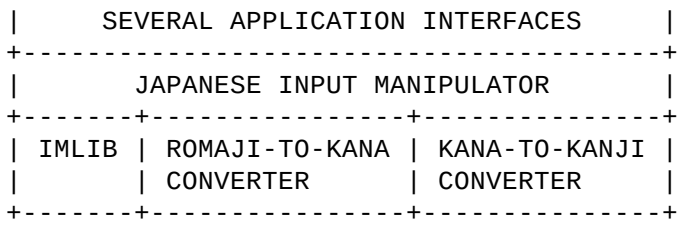
5. HM modifies the information to be displayed according to the action and calls a registered callback routine to update the screen.

HM passes the information that should be displayed on the screen in an argument of the callback routines. The callback routines are prepared by the application and registered into HM context at the initialization of HM. This callback method makes the application interface and data flow more easily.

Components

Figure 9 shows the composition of HM. The application interfaces include both the C and the VMS binding interfaces for the OpenVMS operating system.

Figure 9 HM Component Structure



The Japanese input manipulator performs all Japanese input operation by using IMLIB, the romaji-to-kana converter, and the kana-to-kanji converter. After it processes the input key, it calls back the application routines. There are several types of romaji-to-kana converters. We implemented a submodule romaji-to-kana converter driven by a conversion table; a user can change this table to another.

The kana-to-kanji converter module is a generalized Japanese conversion library. Many Japanese conversion engines exist, and each one is used differently. The kana-to-kanji converter loads the interface routine that absorbs these differences dynamically at the initialization of the HM context. It then processes the conversion request with any engine.

Services

HM provides 17 library entries. In this section, we describe three basic routines: HMInitialize, HMConvert, and HMEndConversion.

- HMInitialize. This routine creates a context for HM. It accepts three callback entries, a user-defined data pointer that would be passed to the callbacks, and an item list for initial information as its arguments.

- HMConvert. This routine sends a key to HM. The key is represented as a 32-bit data (longword) that is generated by a function HMEncodeKey from an escape sequence that the keyboard sends or by a function HMKeysymToKeycode from a keysym of the X Window System. IMLIB interprets the keycode, and HMConvert performs a conversion in accordance with the information. (A summary of what is executed was given in the Mechanism Overview section.)

- HMEndConversion. This routine aborts the conversion and resets an internal status. It is used when the application has to stop the input for a particular reason, for example, if an application issues the cancel request.

Callback Routines

HM requires three callback routines: start_conversion, format_output, and end_conversion. They are used as follows.

- start_conversion. This routine is called when the conversion string input is started. The application memorizes where the cursor is positioned.

- format_output. This routine is called whenever the information to be displayed has been changed. The application updates the screen.

- end_conversion. This routine is called when the input string is determined. As a result, the application takes the string passed in the argument of the last call of format_output into its input buffer.

The user-defined data pointer, one of the arguments for HMInitialize, is always passed to these callbacks. Since HM is not concerned with its contents, the user can put any kind of information into it.

HM is available on the OpenVMS VAX, OpenVMS AXP, ULTRIX, and DEC OSF/1 AXP operating systems. This portability is due to the module's independence from physical I/O. The major client applications working on these operating systems are DECwindows/Motif, Japanese SMG, and the front-end input process.

IMPLEMENTATION OF THE FRONT-END INPUT PROCESSOR

The front-end input process (FIP) for a dumb terminal supports full operations for the Japanese string manipulation. FIP is implemented on the following operating systems: OpenVMS/Japanese/VAX version 5.5-2 or later versions and OpenVMS/Japanese/AXP version 1.0 or later versions.

Full Operation Support

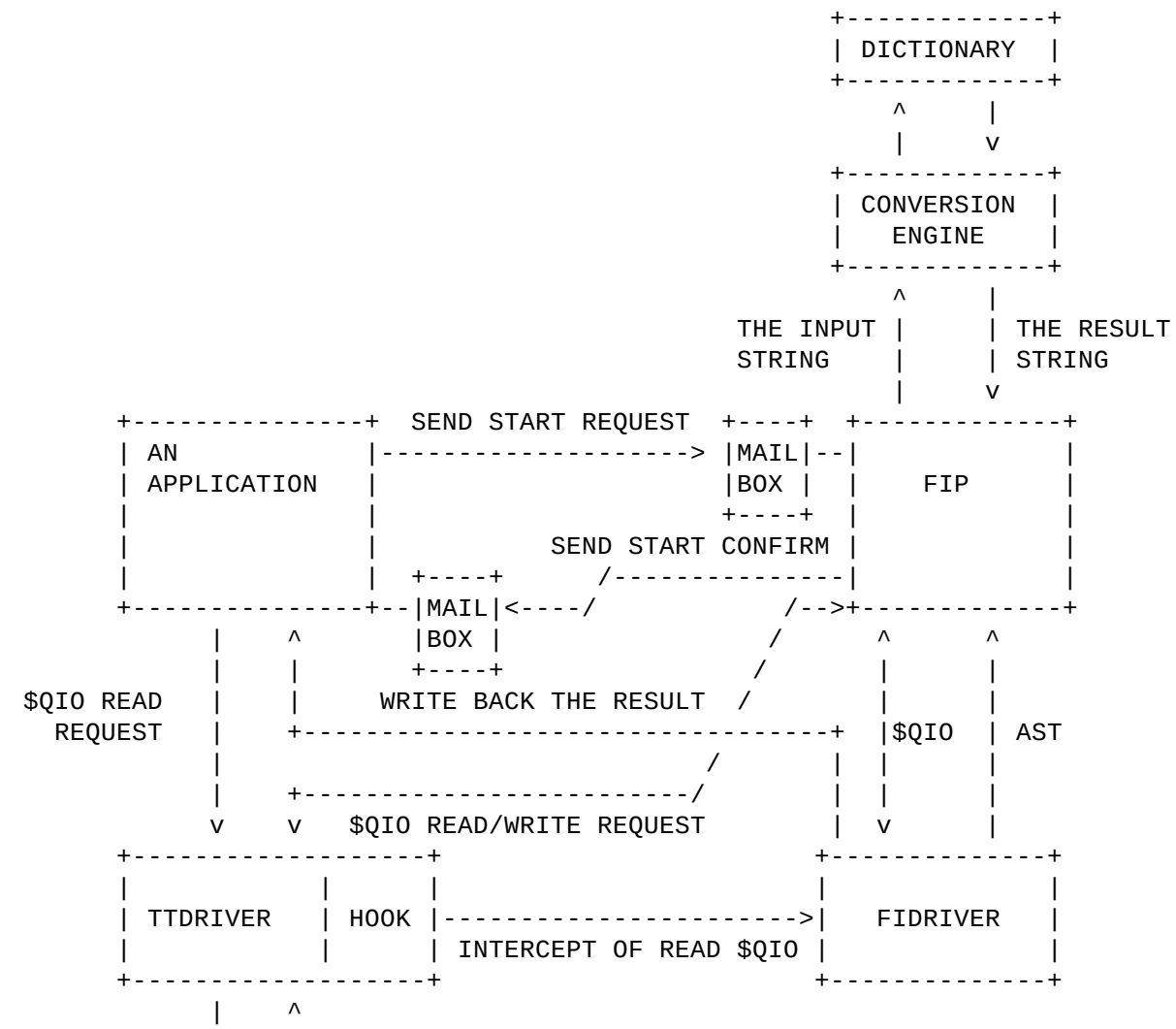
The original product can use FIP if the product's mechanism, particularly its I/O operation and preediting function, does not conflict with the FIP implementation. Some applications conflict with the design of FIP due to the limitations of FIP and its environment. For example, FIP does not detect the read request that includes the NOECHO item code, so the application that issues such a read request to the terminal driver (TTdriver) cannot use FIP as a Japanese front-end input process. Also FIP does not step into a process for the termination of a read request simply because a read buffer that is defined by an application has overflowed. FIP continues to communicate with the TTdriver and a conversion engine to get the Japanese string unless the terminate key is explicitly input. To overcome these conflicts, we implemented a pseudo-driver named FIdriver to intercept I/O requests from the application before they are processed by the TTdriver.

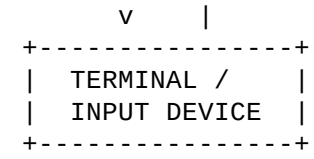
FIP Mechanism Overview

FIP processes all Japanese input operations using HM. We supplied the Digital Command Language (DCL) command, INPUT START/STOP for activating/deactivating FIP. Once a user activates FIP from DCL, it is available until the user logs out or the system is deactivated.

Figure 10 shows FIP and its environment for the manipulation of Japanese input. An application issues I/O requests to the TTdriver to get user inputs, but FIP fetches the requests from the TTdriver through the FIdriver. Then FIP starts to communicate with the drivers and the Japanese string conversion engine to pass the resultant string as well as preedits to a screen.

Figure 10 FIP Environment for Manipulation of Japanese Input





The sequence of the front-end input process follows.

1. An application creates a front-end input process.

2. A front-end input process exchanges packets with an application through its mailbox.

3. An application issues a queued I/O ($QIO) read request to the TTdriver.

4. The FIdriver intercepts the request and passes the information to FIP as a packet.

5. FIP issues a $QIO read request to the TTdriver to get input strings for conversion.

6. A user inputs a key from a terminal. FIP receives the input and decides whether or not to call a routine of the conversion engine. If an input key is recognized as one of the conversion keys, FIP calls the routine and passes the input strings. If not, FIP issues a $QIO write request to the TTdriver to echo an input character.

7. A conversion engine receives a string and converts it to the Japanese string.

8. A conversion engine returns the result to FIP.

9. FIP issues a $QIO write request to the TTdriver to display the resultant string from the engine and arranges the current editing line.

10. Steps 5 to 9 are repeated.

11. Once a user inputs the Terminate key of an application's request, FIP recognizes it as a terminator and returns the entire resultant string to the FIdriver as a write packet.

12. The FIdriver sends the result string and I/O status to an application.

13. An application accepts the converted string. After executing its internal process, it issues another $QIO read request to the TTdriver. (Return to step 3.)

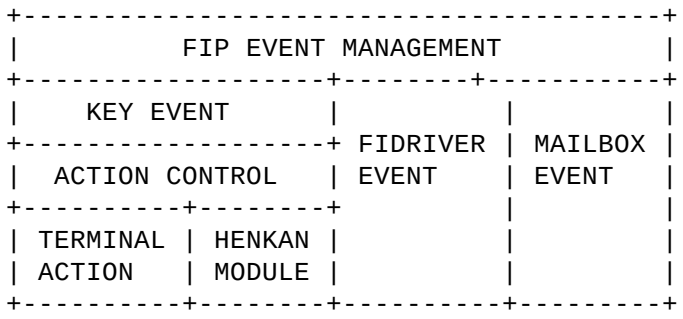
FIdriver. The FIdriver is a pseudo-driver that intercepts $QIO

read requests from an application to the TTdriver. Functioning as a bridge between terminal read requests and FIP, the FIdriver gets a read request, passes its information to FIP, and maintains it. When FIP returns the completion message with its processed Japanese string, the FIdriver validates it and completes a user's read request as if the TTdriver had returned it. Thus the user/application can get the Japanese string without modification for Japanese input method.

The FIdriver has other notification functions for exception handling such as logout, cancel, or abort.

Front-end Input Process Operations. All the operations in the front-end input process are driven by the mailbox event, the FIdriver event, and the key event. Figure 11 shows the functional structure of FIP.

Figure 11 FIP Functional Structure



The following operations in the front-end input process correspond to these three events.

- Mailbox Event. The mailbox event provides communication with an application. FIP issues a read request to its own mailbox. The mailbox event notifies FIP of the arrival of a message from an application. When an application sends a start request to the FIP mailbox, the mailbox event is set so FIP starts to initialize its environment. Also FIP terminates itself at the time a stop request message is delivered to its mailbox.

- FIdriver Event. The FIdriver event provides communication with the FIdriver. The FIdriver intercepts a request from an application to the TTdriver and creates a packet for FIP. FIP issues a read request to the FIdriver, and this event is set when a packet is delivered. A request is categorized in three types: read request, cancel request, and disconnect request.

- Key Event. The key event provides communication with the TTdriver. FIP issues a $QIO read request to the TTdriver

byte by byte. All the input from a keyboard is recognized as a key event in FIP. Once a key event is set in FIP, FIP examines the key sequence in a read buffer.

If the input is in the range of a terminator mask, FIP terminates a read operation from the TTdriver and writes back the resultant string and I/O status block to the FIdriver as a write packet. (A terminator mask is defined in the $QIO read request from an application.)

If the input key is a conversion key, FIP calls a conversion engine and gets the resultant converted string. Then FIP issues a write request to the TTdriver to display the updated string.

If the input key is a printable character, FIP updates the contents of its internal buffers defined in the context and issues a write request to the TTdriver to echo the character.

If the input key is for line editing, for example, to delete a line or a word or to refresh a line, FIP emulates the line-editing function of the TTdriver so its editing function is executed.

FIP stores all user input and read-request information from an application in its internal buffers and database. The buffers contain the codes of user input and corresponding video attributes to display. The database contains item codes in a read request, channel numbers to connect other devices, and so on.

FIP creates a new database when the updated read request from an application is delivered, in other words, when the FIdriver event is set. Also, FIP adds the ASCII code and an attribute of the updated user input into buffers when a user inputs, that is, when the key event is set.

CLIENT/SERVER CONVERSION

The use of a client/server conversion has two advantages: (1) It reduces the required resources for language conversion by distributing some components to other systems, and (2) It presents an environment that shares a common dictionary.

All procedures for the Japanese conversion require large system resources such as CPU power. A user can place the conversion information server (CIserver) and a dictionary on a remote node and call some functions of the CIserver client library to get the resultant string. In this way, a local system saves its resources while the remote server processes the conversions.

In addition, many users can access a common dictionary on the specific remote node. It is possible for any local user to access a dictionary on a remote node if the CIserver on the node is active.

CIserver

The object name is "IM$CISERVER". The CIserver initializes itself by finding the name of a transport protocol in a logical table. It then creates corresponding shareable images, maps its required routines, and waits for a connect request from a client. The CIserver communicates with its client via a mailbox at the transport level. The server sets the asynchronous system trap to the mailbox and reads a message in it such as a connect request, a disconnect request, a connect abortion, or a client's image termination. The CIserver can identify the connection to a client and specify a conversion stream in the connection.

CIserver Client Library. The client library presents programming interfaces. These are callable routines that execute various string manipulations and operations for the Japanese conversion. The CIserver client library is located between an application and the CIserver body.

Input Method Control Program (IMCP). IMCP is a command line interface to customize the CIserver environment. A user sets proxy to a Japanese system dictionary at a remote node on the network, and IMCP administrates a proxy database. A user can confirm the status of the server at a command line and can shut down the server from the IMCP interface.

Other Servers. HM has a conversion engine dispatcher that can dynamically select from several Japanese conversion engines. HM now serves the CIS (CIserver, Digital Japan), the Wnn (Omron Company), the Canna (NEC), and the JSY (Digital Japan) engines. Therefore, an application that uses HM as the Japanese conversion interface can select its preferred engine.

EXTENSION IN THE FUTURE

In this section, we describe the possibilities for internationalization of FIP, HM, IMLIB, the CIserver, and the FIdriver. Although our approach does not provide a multilingual input method, it does provide an architecture that can be used for any language.

FIP has a multibyte I/O operation that can be applied to other two-byte languages. In addition, all the read/write communications among FIP, the FIdriver, and the TTdriver proved able to handle one-byte languages such as English. Also, IMLIB can expand its keybinding system for conversion of other languages, and HM can add the interfaces for conversion engines of other languages if such engines are prepared.

SUMMARY

The Japanese input method is a complex procedure involving preediting operations. Various keybinding systems and manipulators accelerate input operations. Our approach for the Japanese input method allows an application three choices: (1) An application can use a front-end input processor to perform all operations including I/O. (2) An application can use the henkan module and implement I/O operation itself. (3) An application can execute all operations except keybinding, which is handled by an input method library.

ACKNOWLEDGMENTS

We want to express our appreciation to Katsushi Takeuchi of the XTPU development team for his initial designing and prototyping of IMLIB and some implementation of FIP, and Junji Morimitsu on the same team for his initial implementation of IMLIB and its compiler. Also, we wish to thank Makoto Inada on the DECwindows team for his implementation of HM; Hitoshi Izumida, Tsutomu Saito, and Jun Yoshida from the JVMS driver team for their contribution toward creating the FIdriver; and Naoki Okudera for his implementation to the entire CIserver environment. As a final remark, we acknowledge Eiichi Aoki, an engineering manager of ISE Japan, and Hirotaka Yoshioka in the ISA group for their encouragement in writing this paper.

REFERENCES

1. Guides to the X Window System Programmer's Supplement for Release 5 (Sebastopol, CA: O'Reilly & Associates, Inc., 1991).

2. Standard X, Version 11, Release 5 (Cambridge, MA: MIT X Consortium, 1988).

3. K. Yoshimura, T. Hitaka, and S. Yoshida, "Morphological Analysis of Non-marked-off Japanese Sentences by the Least BUNSETSU's Number Method," Transactions of Information Processing Society of Japan, vol. 24 (1983).

4. K. Shirai, Y. Hayashi, Y. Hirata, and J. Kubota, "Database Formulation and Learning Procedure for Kakari-Uke Dependency Analysis," Transactions of Information Processing Society of Japan, vol. 26 (1985).

5. IMLIB/OpenVMS Library Reference Manual (in Japanese) (Tokyo: Digital Equipment Corporation Japan, Order No. AA-PU8TA-TE, 1993).

6. User's Manual for Defining User Keys in IMLIB (in Japanese) (Tokyo: Digital Equipment Corporation Japan, Order No.

AA-PU8UA-TE, 1993).

TRADEMARKS

The following are trademarks of Digital Equipment Corporation: DEC OSF/1 AXP, DECwindows, DECwrite, Digital, OpenVMS AXP, OpenVMS VAX, ULTRIX, and VMS.

Motif and is a registered trademark of the Open Software Foundation, Inc.

Unicode is a trademark of Unicode Inc.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

X Window System is a trademark of the Massachusetts Institute of Technology.

BIOGRAPHIES

Hiroyoshi Baba Hiroyoshi Baba is an engineer in the Japanese Input Method Group in Digital Japan, Research and Development Center. He is currently developing the Japanese front-end input system on OpenVMS VAX and OpenVMS AXP and the Japanese language conversion server system. He received a B.S. (1989) and an M.S. (1991) in electronics engineering from Muroran Institute of Technology, Japan. He joined Digital in April 1991.

Takahide Honma A senior software engineer, Takahide Honma leads the Japanese Input Method Group. He joined Digital in 1985 as a software service engineer. He has worked on systems such as real-time drivers, network system (P.S.I.), and database on VMS and was a consultant to customers. At the same time, he also took the role of a sales advisory support engineer. Since 1990, he has been with Research and Development in Japan and has worked on the Japanese input method. He has an M.S. (1983) in high-energy physics from Kyoto University and is a member of the Physics Society of Japan.

Kuniaki Takizawa Kuniaki Takizawa is an engineer with Digital Japan, Research and Development Center and is a member of the Japanese Input Method Group. He joined Digital in April 1991 and is currently developing and porting the henkan module and the input method library (IMLIB) on OpenVMS, ULTRIX, and OSF/1. He graduated from the University of Electronic Communications (Denki-Tsushin University) in Japan in 1991. His speciality area was the structure of operating systems.

==========