# Section 7

# Development Reference

## Command Documentation Style

This section of the *Reference Manual* offers advice and direction for those users wanting to expand or tailor Humdrum so as to better suit a given application. Two types of extensions are possible. First, the user may define one or more new representation schemes that better represent the types of information of interest. For example, the user might define a new Humdrum representation scheme suitable for representing North Indian *tabla bols*. Second, users might wish to develop new software tools that manipulate one or more Humdrum representations in some fashion. For example, the user might create a command that identifies the roots of chords.

The ensuing section is divided into two parts. The first part *(Representation Development)* provides guidelines for defining new Humdrum representations. The second part *(Software Development)* provides tips for writing adjunct software; a standard *program skeleton* is described.

# 1. Humdrum Representation Development

## 1.1 Representation Assumptions

Three principle assumptions underly the Humdrum syntax. As with all design assumptions, these principles inevitably act as limitations — circumscribing what is possible.

The first assumption is that the information can be adequately represented using discrete rather than continuous signifiers. This is a limitation of all *symbolic* as opposed to *analogic* representations. If continuous data (such as conducting gestures, or analog sound) are to be represented using Humdrum, the data must be somehow transformed into a discrete form.

The second assumption is that information can be meaningfully organized as ordered successions of data tokens. More concretely, it is assumed that data can be arranged in linear *spines*, and that these spines are interpretable. This limitation implies that data tokens can be meaningfully ordered. Normally, *time* (as in sequences of events), or *space* (as in successive printed signs) provide suitable ordering devices. However, events that are conceived as entirely independent of one another are difficult to represent in Humdrum.

The third assumption is that the ASCII character set provides sufficient richness to act as an appropriate set of signifiers. In practice, this third assumption proves to be the most limiting. The use of alphabetic and numeric characters is itself of little concern; as Saussure pointed out, the choice of symbols is arbitary and conventional. Of greater concern is the limitation of size. There are some 128 characters in the basic ASCII set — not all of them are printable.

The size restriction imposed by the ASCII character set might be circumvented by using binary, pictoral or other signifiers. However, there are currently significant advantages to using ASCII signifiers. The principal advantage is that many important software tools already exist for the manipulation of ASCII text. The principal disadvantage of ASCII is addressed in Humdrum by allowing users to recycle the ASCII characters an indefinite number of times — through the use of new exclusive interpretations. Note that if all representation schemes use ASCII signifiers, then any software tool developed for the manipulation of ASCII text can be applied immediately to any Humdrum representation. This approach discourages the explosive proliferation of specialized software (each dealing with a unique representation), and encourages users to rely on a smaller toolkit consisting of more generalized and flexible tools. More precisely, this approach reduces the demands for software development while maximizing the range of tasks to which a user's skills may be applied.

An alternative to Humdrum's provision for multiple representations would be to provide a single representation, with greater contextual constraints on the positioning of signifiers. Unfortunately, high levels of context dependency end up placing an inordinate burden on software development and use. Humdrum attempts to simplify software development by minimizing as much as possible context-dependent meanings for signifiers.

## 1.2 Creating New Humdrum Representations

The Humdrum syntax provides a framework within which different music-related symbol-systems can be defined. Each symbol system or representation *scheme* is denoted by a unique *exclusive*

*interpretation.* There is no restriction on the number of schemes that can be used or created within Humdrum. Humdrum users are consequently free to design new representation schemes that address various needs.

Many newly created representation schemes are apt to be *user-specific* — ways of organizing or representing information that are not likely to be of much value to other Humdrum users. In other cases, a well crafted representation scheme will prove to be of benefit to a wider community of users. Whether a new Humdrum representation scheme is intended for private use or for public distribution, it is prudent to develope good design habits. There is no such thing as a "perfect" representation, but it is possible to distinguish poor representations from better representations.

There are a number of considerations involved in the lucid design of a Humdrum interpretation. In general, the procedures involved in representation design can be summarized as follows:

1. Identify as clearly as possible the goal or goals you hope to achieve using this new representation. Formulate a list of questions you expect to be able to answer or address.
2. Create a list of the *essential signifieds* (concepts to be represented) that are needed to pursue the goal(s).
3. Make a supplementary list of *related signifieds* that seem peripheral to the immediate goal(s), but might prove important in pursuing related goals.
4. By examining the lists of essential and related signifieds, try to identify and define the *class* or *classes* of information with which you are dealing.
5. Identify whether the signifieds of immediate interest belong together in a single representation, or whether they ought to be split into two or more Humdrum interpretations.
6. Having established a clearer understanding of the class of signifieds, trim the previous lists of signifieds to a single list.
7. Identify those signifieds that may take many variant forms (as, for example, musical ornament symbols). Determine whether these variant forms are *finite, infinite* or *unknown* in number. If the number is infinite or unknown, a single signifier should be used — with provision for an auxilliary representation that can be used to identify the specific variant form.
8. Assign signifiers to all signifieds using the supply of available ASCII characters. In general, assign only a single character for each signified. Using individual ASCII characters as signifiers helps eliminates context dependency, and so reduces the complexity of the software needed to process the representation.
9. In assigning signifiers to signifieds, try to avoid English language initialisms (such as Q for quarter-note), since these mappings are poor mnemonics for non-English-speaking users.

Some of the "do's" and "dont's" of designing a Humdrum interpretation can be illustrated by considering a hypothetical representation problem. Below, we consider how to design a representation for keyboard fingering.

## 1.3 Defining a New Humdrum Interpretation: A Sample Problem

Suppose that we are interested in providing a comprehensive representation scheme for fingering keyboard instruments. More concretely, let's suppose that we are interested in studying the degree of performance difficulty for various keyboard works. Our *goal* might be to compare the relative difficulty of works by different composers, or whether a given musical arrangement is easier to perform than another arrangement. We might imagine, for example, writing a program which, given some knowledge of performance constraints, could accept fingering information as input and produce as output an index of the degree of difficulty. At a later stage, we might imagine creating an "intelligent" fingering program that could be used to determine an optimum way of fingering a particular keyboard passage. (Such a program might even be designed to take into account the unique physiological abilities, constraints, or preferences of a given performer.) In summary, our first research goal would use our "fingering" representation as an input to some analysis program, whereas our second research goal might produce the "fingering" representation as output.

Having identified the above goals, our next task is to identify the *essential signifieds* that would be needed to solve these problems. The most basic information we want to represent includes:

1. The identity of the finger and hand used in each key-press.
2. The identify of the key used in each key-press.
3. The order or sequence of key-presses.
4. The timing of each activity or movement.

Having identified what we see as the essential signifieds, we ought to pause and consider related signifieds that, although they appear to be peripheral to our goals, might prove important in pursuing related goals. By thinking ahead about these other signifieds, we might avoid future difficulties should we discover that another item of information proves crucial to our enterprise.

Some potential properties or attributes that we might consider representing could include the following:

1. The force or velocity with which the key is pressed.
2. Whether the hands are crossed — and if so, which arm is placed above the other.
3. What part of the finger/hand is used to press the key (e.g. knuckles).
4. Whether more than one finger is used to press the same key together.
5. Whether one (or more) finger is substituted for another finger in the course of holding a depressed key.
6. Whether trills are notated as a precise sequence of finger presses, or whether they are recorded as generic "trills."
7. Whether "Bebung" is used — that is, whether lateral or vertical pressure is applied once the key is depressed.
8. Whether a second performer can be accommodated (as in the case of a piano duet).

9. Physiological or anatomical attributes of the performer (such as hand-spans).
10. Pedalling.

A list of related signifieds is rarely likely to be exhaustive or complete. So it is important to take time to consider other possible types of information that relate to keyboard fingering.

In the initial stages of designing a Humdrum representation, it is generally wise to try to formulate a fairly exhaustive list of possible pertinent attributes. The purpose of such a list is to ensure that an informed decision is made regarding those properties we wish to include in the representation, and those properties we propose to exclude. More specifically, our goal is to exclude information on the basis of an explicit decision rather than due to a tacit oversight.

Given the above list of potential signifieds, the next step is to pause and consider the nature of the *class* of information we wish to deal with. For example, the idea of "pedalling" raises an interesting representation question. Are we trying to represent keyboard *fingering?* Or is our task the representation of keyboard *performance*? Pedalling is obviously part of keyboard performance, but not something fingers do. Are we mistaken in thinking that our representation task is limited to fingering? Also, since larger arm and body movements are essential aspects of good performance, should we also consider representing these additional factors?

In light of our goal of measuring performance difficulty, we would have to admit that pedalling can indeed contribute to the physical challenge arising from performing a given work. In terms of our research task therefore, it makes sense to include pedalling information. However, we might balk at the prospect of mixing fingering and pedalling within a single representation — especially since some keyboard instruments (e.g. clavichord) have no pedals. Moreover, the pedals on a piano differ considerably from the pedals on an organ, although both contribute to overall performance difficulty.

At this point, we are invited to consider whether the various signifieds in the above list truly belong together in a single *representation*, or whether they ought to be split into two or more Humdrum interpretations. The above discussion suggests that we might distinguish at least five classes of information: *performance (broadly construed), fingering, body movement, pedal-boarding* (as on the organ), and *pedalling* (as on the piano or harpsichord). Moreover, we might define "performance information" as the combination of fingering plus pedalling or pedal-boarding. In short, it would make sense to define *three* representations: fingering, pedalling, and pedal-boarding, and to assume that our program measuring performance difficulty will accept any combination of one or more of these three classes of information.

Our task has clearly expanded somewhat, since now we need to consider three types of representation rather than one. For the purposes of this tutorial example, we might set aside the problems of representing pedalling and pedal-boarding and focus on the fingering aspect of keyboard performance.

Now that we have a clearer understanding of the class of signifieds, we can begin to trim the lists of signifieds to a single short list. We have decided not to represent pedalling using the same Humdrum interpretation as for fingering. We might also decide that the fingering activity for a second performer can be represented using a second independent spine of information. We might also dispense with representing the force or velocity of key-depression, and what part of the finger/hand is used to press the key. We might have decided to represent the fingering for trills, but not to encode each key-stroke of the trill separately. We could also decide that representing physiological attributes of the performer (such as hand-spans) ought to be left as a

separate representation. Finally, we might have also decided not to represent *Bebung*. This leaves a trimmed list of eight types of signifieds:-

1. The identity of the finger and hand used in each key-press.
2. The identify of the key used in each key-press.
3. The order or sequence of key-presses.
4. The duration of each activity or movement.
5. Whether the hands are crossed — and if so, which arm is placed above the other.
6. Whether more than one finger is used to press the same key together.
7. Whether finger substitution occurs.
8. The fingering for trills.

Before going on to map signifiers and signifieds we need to consider those signifieds that may take variant forms. What sort of "variations" might appear in a fingering representation? An obvious form of variation occurs when alternative fingerings are possible — that is, where a passage contains two (or more) ways of assigning key-presses to different fingers. Our first task here is to determine whether the number of variant forms is finite, infinite, or unknown in number. We can consider this question both at the level of the individual key-press, and at the level of the entire work. In the case of the individual key-press, the maximum number of variants is ten — since there are no more than ten fingers. If more than one finger is used to press a key, or if finger-substitutions occur, then the maximum number of variants is somewhat more than ten — although still finite in number. At the level of the entire work, the maximum number of variants is potentially very large (at least 10 × the total number of key-presses in the work). Nevertheless, this number remains a finite value for works of finite length. The question arises, do we want our fingering representation to represent a single performance of a keyboard passage, or is the representation intended to represent alternative forms of performance?

In order to answer this question we must return once again to our initial goals. In analysing the degree of performance difficulty for a work, we might prefer to analyse a single (actual or plausible) performance. Measuring the degree of performance difficulty for a *class* of variant performances is apt to prove difficult. On the other hand, the musical score for a keyboard work may contain no fingering indications whatsoever. Therefore it would be wrong to assume that a single fingering specification would give an accurate indication of the performance difficulty for a given musical work. This raises the question of whether our intention is to measure the performance difficulty of a specific sequence of key-presses used in a performance, or whether our intention is to measure the performance difficulty of a particular musical work.

As noted above, measuring the difficulty for a class of variant performances is likely to prove difficult. There are many many ways of fingering a keyboard work. Averaging the performance difficulty for the complete class of possible fingering arrangements would appear silly since most of these fingerings would be awkward. One could argue, for example, that the degree of performance difficulty for a work can be best established by analysing the single most convenient way of fingering the work. Alternatively, one could argue that the degree of performance difficulty for a work can be determined by examining a handful of the most convenient ways of fingering the work. Out of the large number of *possible* fingerings, it is only the *plausible* fingerings that really count.

Whether our intention is to measure a single fingering sequence or a class of such fingerings, a helpful question to consider here is where do we expect to get our fingering data? Three sources come to mind: (1) recorded fingering data from an actual performance, (2) fingerings (including alternatives) notated in printed scores or annotated by keyboard performers, and (3) fingering data (including alternatives) generated by a computer program. In the case of an actual performance, there will be only one fingering. The other sources can potentially produce more than one fingering at a time.

Having identified those attributes that we wish to represent, we need to consider how the representation ought to be structured. Specifically, we need to consider how our fingering representation can be coordinated with other Humdrum interpretations. The most important coordination task is ensuring that our representation will correspond well with the core `**kern` representation. Each data record in `**kern` represents a single sonority — a moment in time that differs from the previous state. Since key-presses are closely related to notes, we might want to coordinate each of the `**kern` note tokens with possible key-presses. Many Humdrum pitch-related representations include barlines — which are useful markers for coordinating such representations. This suggests that it might be useful to include barlines in our fingering representation. Given both the barline and note-token/key-press correspondences, we should have little difficulty ensuring that our fingering representation will be fully coordinated with a number of other Humdrum representations.

We are now ready to consider how to map our signifieds with a set of appropriate signifiers. In general, we should endeavor to define one signifier for each attribute. First, consider how we might identify the individual fingers. A good system would be to identify each finger by a unique signifier — such as a unique decimal integer. However, there is a long-standing tradition of identifying the thumb of each hand as the number "1", the index finger by the number "2" and so on. Given the limited number of fingers, some context-dependency may be appropriate here. In short, we may decide to identify specific fingers through a *ligature* of "hand+finger" — e.g. left-3 or right-5. The signifiers "left" and "right" obviously introduce an English bias. It would be better to consider more universally recognized terms such as "mano destra" (MD) and "mano sinistra" (MS). However, not every user will find these is terms familiar or comfortable.

We need not rely on a literalism or initialism. "Left" and "right" are concepts that lend themselves well to pictorial representation, so we might consider using those ASCII characters that convey a left-right pictorial dimension. Possible contenders would include various letter-contrasts: `d` versus `b`, `J` versus `L`; the three types of parentheses: `(` versus `)`, `{` versus `}`, and `[` versus `]`; and the greater-than and less-than signs: `<` versus `>`. The letter contrasts `J` versus `L` are especially poor since although the angle of the letter `L` is drawn to the right, "L" implies an initialism for "left" — and so is apt to cause confusion. The parentheses cannot be misconstrued as literalisms or initialisms, so they are somewhat better signifiers. However the greater-than and less-than signs are the mostly clearly arrow-like, and so perhaps provide a better pair of left-right signifiers.

Having decided upon the signifiers for left and right, and having adopted the tradition of numbering the fingers (1=thumb, 2=index, 3=middle, 4=ring, 5=little), we could continue mapping signifiers to signifieds, taking care to minimize context dependency.

In reflecting on the above discussion, readers are apt to feel that one or another type of information ought to have been included, or that the signifiers ought to be assigned in a different manner. Since Humdrum provides a framework within which alternative representation schemes

can be designed, there is no need to defend a given representation from competing schemes. Whether an interpretation survives and proliferates will be determined, not by its conceptual elegance or completeness, but by whether it is found to have a practical utility in solving users' problems.

## 2. Humdrum Software Development

The Humdrum toolkit is necessarily limited in scope and there are many functions that users will wish to add. In developing adjunct software tools, it is imperative that the software conform to the following design conventions:

1. Programs should be general-purpose and adapt to a wide variety of input circumstances.
2. If possible, programs should be able to process any Humdrum input rather than be limited to a given type of input interpretation.
3. Command names should be limited to 8 characters in length in order to ensure portability to DOS systems.
4. Command names should preferably be the same as the output interpretation produced by the command.
5. Command names should not be unduly abbreviated since infrequently used software is less easily remembered than frequently used system commands.
6. The command syntax should conform to standard POSIX conventions.
7. Errors and warnings should be prefaced by giving the name of the program or command which issues the error message. e.g.

```
vox: ERROR: voice 3 begins with a null token.
```

8. Errors messages should be sent to "stderr" rather than to the standard output.
9. Wherever possible, 'filter' programs should produce outputs that are identical in structure to the input. More specifically, input line numbers should correspond to output line numbers — where appropriate.
10. Comments, interpretations, barlines, and double barlines should be echoed in the output as the default condition (except in the case of formatted non-Humdrum outputs).
11. For many programs, the user should be able to skip the processing of certain types of tokens (such as barlines) by specifying a **-s** flag — followed by a user-defined regular expression. Tokens matching the regular expression should be echoed unprocessed in the output stream.
12. Programs should handle spine-path changes in a fashion appropriate to the nature of the command.
13. Comments and interpretations should be identified by explicitly matching the exclamation mark or asterisk in the *first* column of the input data token. Exclamation marks and asterisks are legitimate data signifiers when not occurring in the first column of an input token.
14. Where possible, outputs should *not* be formatted with descriptive labels etc. The preferred output format is to have all outputs conform to the Humdrum syntax. This ensures that all outputs can themselves be used as inputs to other Humdrum programs.
15. Programs should generally avoid assumptions concerning context-dependent inputs. Inputs should be assumed to be context-free.

16. Programs should be able to handle inputs with unexpected user extensions or representational addenda — such as the presence of spurious or unknown characters.

17. Programs that search or examine inputs for certain features, properties, or errors should return a *null* output if nothing is found. Messages indicating that 'nothing was found' should be avoided. "Silence is golden."

## 2.1 Standard Program Skeleton

Much of the Humdrum software was originally developed using the AWK programming language. AWK was designed by Alfred Aho, Brian Kernighan, and Peter Weinberger.† It is syntactically very similar to the C programming language, but is easier to use and promotes better software productivity. AWK provides powerful text manipulation features that make it admirably suited to the creation of Humdrum software. AWK is also a very easy language to learn, and is an excellent first language for novice programmers.

The Humdrum Toolkit includes programing skeletons that may provide a useful starting place for software development using AWK. Two skeleton files are provided with the toolkit: `skeleton.ksh` and `skeleton.awk`. The kornshell file (`.ksh`) parses the command line, issues appropriate error messages if the command is improperly invoked, displays a help screen if necessary, and assembles the command parameters to invoke an awk script for the command (`.awk`).

The `skeleton.awk` skeleton contains a main loop that is normally executed for each record of input. A series of useful functions are included in the AWK skeleton program. These functions include:

**Parse_command.** This function checks that the input passed from the corresponding kornshell script for the command. The Parse_command function contains a list of valid options and assigns the passed parameters to the appropriate option variables.

**Store_indicators.** This function allows the spine-path indicators for the current record to be stored in the array `path_indicator` so that they may be used later.

**Store_new_interps.** This function stores the new interpretations found in an interpretation record for each spine.

**Process_indicators.** This function takes the spine-path indicators that were stored in the array 'path_indicator' in the function 'store_indicators' and manipulates the arrays 'path_indicator' and 'current_interp' according to the contents of the array 'path_indicator'.

**Ins_array_pos.** This function inserts new positions in the arrays 'path_indicator', 'current_interp', and 'current_key' and copies elements so that everything is preserved

**Del_array_pos.** Performs the opposite of function 'ins_array_pos'.

**Exchange_spines.** This function exhanges two spines by exchanging the corresponding elements in current_interp.

---

† Aho, A., Kernighan, B. & Weinberger, P. *The AWK Programming Langauge*. Reading, Massachusetts: Addison-Wesley Publishing Co. 1988.

Zwicker, E., Flottorp, G. & Stevens, S. S.
"Critical bandwidth in loudness summation," *Journal of the Acoustical Society of America*, Vol. 29, No. 5 (1957) pp. 548-557.